

My Collection



Developer Network

This document is provided "as-is". Information and views expressed in this document, including URL and other Internet Web site references, may change without notice. This document does not provide you with any legal rights to any intellectual property in any Microsoft product or product name. You may copy and use this document for your internal, reference purposes. You may modify this document for your internal, reference purposes. © 2013 Microsoft. All rights reserved.
Terms of Use (<http://msdn.microsoft.com/cc300389.aspx>) | Trademarks (<http://www.microsoft.com/library/toolbar/3.0/trademarks/en-us.msp>)

Table Of Contents

Framework Design Guidelines

[Design Guidelines for Class Library Developers](#)
[Naming Guidelines](#)
[Class Member Usage Guidelines](#)
[Type Usage Guidelines](#)
[Guidelines for Exposing Functionality to COM](#)
[Error Raising and Handling Guidelines](#)
[Array Usage Guidelines](#)
[Operator Overloading Usage Guidelines](#)
[Guidelines for Casting Types](#)
[Common Design Patterns](#)
[Security in Class Libraries](#)
[Threading Design Guidelines](#)
[Guidelines for Asynchronous Programming](#)
[Framework Design Guidelines](#)

Design Guidelines for Class Library Developers

.NET Framework 1.1

The .NET Framework's managed environment allows developers to improve their programming model to support a wide range of functionality. The goal of the .NET Framework design guidelines is to encourage consistency and predictability in public APIs while enabling Web and cross-language integration. It is strongly recommended that you follow these design guidelines when developing classes and components that extend the .NET Framework. Inconsistent design adversely affects developer productivity. Development tools and add-ins can turn some of these guidelines into de facto prescriptive rules, and reduce the value of nonconforming components. Nonconforming components will function, but not to their full potential.

These guidelines are intended to help class library designers understand the trade-offs between different solutions. There might be situations where good library design requires that you violate these design guidelines. Such cases should be rare, and it is important that you provide a solid justification for your decision. The section provides naming and usage guidelines for types in the .NET Framework as well as guidelines for implementing common design patterns.

In This Section

[Relationship to the Common Type System and the Common Language Specification](#)

Describes the role of the common type system and the Common Language Specification in class library development.

[Naming Guidelines](#)

Describes the guidelines for naming types in class libraries.

[Class Member Usage Guidelines](#)

Describes the guidelines for using properties, events, methods, constructors, fields, and parameters in class libraries.

[Type Usage Guidelines](#)

Describes the guidelines for using classes, value types, delegates, attributes, and nested types in class libraries.

[Guidelines for Exposing Functionality to COM](#)

Describes the guidelines for exposing class library types to COM.

[Error Raising and Handling Guidelines](#)

Describes the guidelines for raising and handling errors in class libraries.

[Array Usage Guidelines](#)

Describes the guidelines for using arrays in class libraries and how to decide whether to use an array vs. a collection.

[Operator Overloading Usage Guidelines](#)

Describes the guidelines for implementing operator overloading in base class libraries.

[Guidelines for Implementing Equals and the Equality Operator \(==\)](#)

Describes the guidelines for implementing the **Equals** method and the equality operator (==) in class libraries.

[Guidelines for Casting Types](#)

Describes the guidelines for casting types in class libraries.

[Common Design Patterns](#)

Describes how to implement design patterns for **Finalize** and **Dispose** methods, the **Equals** method, callback functions, and time-outs.

[Security in Class Libraries](#)

Describes the precautions to take when writing highly trusted class library code, and how to help protect resources with permissions.

[Threading Design Guidelines](#)

Describes the guidelines for implementing threading in class libraries.

[Guidelines for Asynchronous Programming](#)

Describes the guidelines for implementing asynchronous programming in class libraries and provides an asynchronous design pattern.

Related Sections

[Class Library](#)

Documents each of the public classes that constitute the .NET Framework.

Relationship to the Common Type System and the Common Language Specification

.NET Framework 1.1

The [Common Type System](#) is the model that defines the rules the common language runtime follows when declaring, using, and managing types. The common type system establishes a framework that enables cross-language integration, type safety, and high-performance code execution. It is the raw material from which you can build class libraries.

The Common Language Specification (CLS) defines a set of programmatically verifiable rules that governs the interoperation of types authored in different programming languages. Targeting the CLS is an excellent way to ensure cross-language interoperation. Managed class library designers can use the CLS to guarantee that their APIs are callable from a wide range of programming languages. Note that although the CLS encourages good library design, it does not enforce it. For more information on this topic, see [Writing CLS-Compliant Code](#).

You should follow two guiding principles with respect to the CLS when determining which features to include in your class library:

1. Determine whether the feature facilitates the type of API development appropriate to the managed space.
The CLS should be rich enough to provide the ability to write any managed library. However, if you provide multiple ways to perform the same task, you can confuse users of your class library about correct design and usage. For example, providing both safe and unsafe constructs forces users to decide which to use. Therefore, the CLS encourages the correct usage by offering only type-safe constructs.
2. Determine whether it is difficult for a compiler to expose the feature.
All programming languages will require some modification in order to target the runtime and the common type system. However, in order for developers to make a language CLS-compliant, they should not have to create a large amount of additional work. The goal of the CLS is to be as small as possible while offering a rich set of data types and features.

See Also

[Common Type System](#) | [Class Library](#) | [What is the Common Language Specification?](#) | [Writing CLS-Compliant Code](#)

Naming Guidelines

.NET Framework 1.1

A consistent naming pattern is one of the most important elements of predictability and discoverability in a managed class library. Widespread use and understanding of these naming guidelines should eliminate many of the most common user questions. This topic provides naming guidelines for the .NET Framework types. For each type, you should also take note of some general rules with respect to capitalization styles, case sensitivity and word choice.

In This Section

[Capitalization Styles](#)

Describes the Pascal case, camel case, and uppercase capitalization styles to use to name identifiers in class libraries.

[Case Sensitivity](#)

Describes the case sensitivity guidelines to follow when naming identifiers in class libraries.

[Abbreviations](#)

Describes the guidelines for using abbreviations in type names.

[Word Choice](#)

Lists the keywords to avoid using in type names.

[Avoiding Type Name Confusion](#)

Describes how to avoid using language-specific terminology in order to avoid type name confusion.

[Namespace Naming Guidelines](#)

Describes the guidelines to follow when naming namespaces.

[Class Naming Guidelines](#)

Describes the guidelines to follow when naming classes.

[Interface Naming Guidelines](#)

Describes the guidelines to follow when naming interfaces.

[Attribute Naming Guidelines](#)

Describes the correct way to name an attribute using the Attribute suffix.

[Enumeration Type Naming Guidelines](#)

Describes the guidelines to follow when naming enumerations.

[Static Field Naming Guidelines](#)

Describes the guidelines to follow when naming static fields.

[Parameter Naming Guidelines](#)

Describes the guidelines to follow when naming parameters.

[Method Naming Guidelines](#)

Describes the guidelines to follow when naming methods.

[Property Naming Guidelines](#)

Describes the guidelines to follow when naming properties.

[Event Naming Guidelines](#)

Describes the guidelines to follow when naming events.

Related Sections

[Class Member Usage Guidelines](#)

Describes the guidelines for using properties, events, methods, constructors, fields, and parameters in class libraries.

[Type Usage Guidelines](#)

Describes the guidelines for using classes, value types, delegates, attributes, and nested types in class libraries.

[Design Guidelines for Class Library Developers](#)

Provides naming and usage guidelines for types in the .NET Framework as well as guidelines for implementing common design patterns.

Capitalization Styles

.NET Framework 1.1

Use the following three conventions for capitalizing identifiers.

Pascal case

The first letter in the identifier and the first letter of each subsequent concatenated word are capitalized. You can use Pascal case for identifiers of three or more characters. For example:

```
BackCol or
```

Camel case

The first letter of an identifier is lowercase and the first letter of each subsequent concatenated word is capitalized. For example:

```
backCol or
```

Uppercase

All letters in the identifier are capitalized. Use this convention only for identifiers that consist of two or fewer letters. For example:

```
System. IO  
System. Web. UI
```

You might also have to capitalize identifiers to maintain compatibility with existing, unmanaged symbol schemes, where all uppercase characters are often used for enumerations and constant values. In general, these symbols should not be visible outside of the assembly that uses them.

The following table summarizes the capitalization rules and provides examples for the different types of identifiers.

Identifier	Case	Example
Class	Pascal	AppDomain
Enum type	Pascal	ErrorLevel
Enum values	Pascal	FatalError
Event	Pascal	ValueChanged
Exception class	Pascal	WebException Note Always ends with the suffix Exception .
Read-only Static field	Pascal	RedValue
Interface	Pascal	IDisposable Note Always begins with the prefix I .
Method	Pascal	ToString
Namespace	Pascal	System.Drawing
Parameter	Camel	typeName
Property	Pascal	BackColor
Protected instance field	Camel	redValue Note Rarely used. A property is preferable to using a protected instance field.
Public instance field	Pascal	RedValue Note Rarely used. A property is preferable to using a public instance field.

See Also

[Design Guidelines for Class Library Developers](#) | [Naming Guidelines](#)

Case Sensitivity

.NET Framework 1.1

To avoid confusion and guarantee cross-language interoperability, follow these rules regarding the use of case sensitivity:

- Do not use names that require case sensitivity. Components must be fully usable from both case-sensitive and case-insensitive languages. Case-insensitive languages cannot distinguish between two names within the same context that differ only by case. Therefore, you must avoid this situation in the components or classes that you create.
- Do not create two namespaces with names that differ only by case. For example, a case insensitive language cannot distinguish between the following two namespace declarations.

```
namespace ee.cummings;  
namespace Ee.Cummings;
```

- Do not create a function with parameter names that differ only by case. The following example is incorrect.

```
void MyFunction(string a, string A)
```

- Do not create a namespace with type names that differ only by case. In the following example, `Point p` and `POINT p` are inappropriate type names because they differ only by case.

```
System.Windows.Forms.Point p  
System.Windows.Forms.POINT p
```

- Do not create a type with property names that differ only by case. In the following example, `int Color` and `int COLOR` are inappropriate property names because they differ only by case.

```
int Color {get, set}  
int COLOR {get, set}
```

- Do not create a type with method names that differ only by case. In the following example, `calculate` and `Calculate` are inappropriate method names because they differ only by case.

```
void calculate()  
void Calculate()
```

See Also

[Design Guidelines for Class Library Developers](#) | [Naming Guidelines](#)

Abbreviations

.NET Framework 1.1

To avoid confusion and guarantee cross-language interoperability, follow these rules regarding the use of abbreviations:

- Do not use abbreviations or contractions as parts of identifier names. For example, use `GetWindow` instead of `GetWin`.
- Do not use acronyms that are not generally accepted in the computing field.
- Where appropriate, use well-known acronyms to replace lengthy phrase names. For example, use `UI` for User Interface and `OLAP` for On-line Analytical Processing.
- When using acronyms, use Pascal case or camel case for acronyms more than two characters long. For example, use `HtmlButton` or `htmlButton`. However, you should capitalize acronyms that consist of only two characters, such as `System.IO` instead of `System.Io`.
- Do not use abbreviations in identifiers or parameter names. If you must use abbreviations, use **camel case** for abbreviations that consist of more than two characters, even if this contradicts the standard abbreviation of the word.

See Also

[Design Guidelines for Class Library Developers](#) | [Naming Guidelines](#)

Word Choice

.NET Framework 1.1

Avoid using class names that duplicate commonly used .NET Framework namespaces. For example, do not use any of the following names as a class name: **System**, **Collections**, **Forms**, or **UI**. See the [Class Library](#) for a list of .NET Framework namespaces.

In addition, avoid using identifiers that conflict with the following keywords.

AddHandler	AddressOf	Alias	And	Ansi
As	Assembly	Auto	Base	Boolean
ByRef	Byte	ByVal	Call	Case
Catch	CBool	CByte	CChar	CDate
CDec	CDbI	Char	CInt	Class
CLng	CObj	Const	CShort	CSng
CStr	CType	Date	Decimal	Declare
Default	Delegate	Dim	Do	Double
Each	Else	Elseif	End	Enum
Erase	Error	Event	Exit	ExternalSource
False	Finalize	Finally	Float	For
Friend	Function	Get	GetType	Goto
Handles	If	Implements	Imports	In
Inherits	Integer	Interface	Is	Let
Lib	Like	Long	Loop	Me
Mod	Module	MustInherit	MustOverride	MyBase
MyClass	Namespace	New	Next	Not
Nothing	NotInheritable	NotOverridable	Object	On
Option	Optional	Or	Overloads	Overridable
Overrides	ParamArray	Preserve	Private	Property
Protected	Public	RaiseEvent	ReadOnly	ReDim
Region	REM	RemoveHandler	Resume	Return
Select	Set	Shadows	Shared	Short
Single	Static	Step	Stop	String
Structure	Sub	SyncLock	Then	Throw
To	True	Try	TypeOf	Unicode
Until	volatile	When	While	With
WithEvents	WriteOnly	Xor	eval	extends
instanceof	package	var		

See Also
[Design Guidelines for Class Library Developers](#) | [Naming Guidelines](#)

Avoiding Type Name Confusion

.NET Framework 1.1

Different programming languages use different terms to identify the fundamental managed types. Class library designers must avoid using language-specific terminology. Follow the rules described in this section to avoid type name confusion.

Use names that describe a type's meaning rather than names that describe the type. In the rare case that a parameter has no semantic meaning beyond its type, use a generic name. For example, a class that supports writing a variety of data types into a stream might have the following methods.

VB

```
Sub Write(value As Double);
Sub Write(value As Single);
Sub Write(value As Long);
Sub Write(value As Integer);
Sub Write(value As Short);
[C#]
void Write(double value);
void Write(float value);
void Write(long value);
void Write(int value);
void Write(short value);
```

Do not create language-specific method names, as in the following example.

VB

```
Sub Write(doubleValue As Double);
Sub Write(singleValue As Single);
Sub Write(longValue As Long);
Sub Write(integerValue As Integer);
Sub Write(shortValue As Short);
[C#]
void Write(double doubleValue);
void Write(float floatValue);
void Write(long longValue);
void Write(int intValue);
void Write(short shortValue);
```

In the extremely rare case that it is necessary to create a uniquely named method for each fundamental data type, use a universal type name. The following table lists fundamental data type names and their universal substitutions.

C# type name	Visual Basic type name	JScript type name	Visual C++ type name	llasm.exe representation	Universal type name
sbyte	SByte	sByte	char	int8	SByte
byte	Byte	byte	unsigned char	unsigned int8	Byte
short	Short	short	short	int16	Int16
ushort	UInt16	ushort	unsigned short	unsigned int16	UInt16
int	Integer	int	int	int32	Int32
uint	UInt32	uint	unsigned int	unsigned int32	UInt32
long	Long	long	__int64	int64	Int64
ulong	UInt64	ulong	unsigned __int64	unsigned int64	UInt64
float	Single	float	float	float32	Single
double	Double	double	double	float64	Double
bool	Boolean	boolean	bool	bool	Boolean
char	Char	char	wchar_t	char	Char
string	String	string	String	string	String
object	Object	object	Object	object	Object

For example, a class that supports reading a variety of data types from a stream might have the following methods.

VB

```
ReadDouble()As Double
ReadSingle()As Single
ReadInt64()As Long
```

```
ReadInt32() As Integer
ReadInt16() As Short
[C#]
double ReadDouble();
float ReadSingle();
long ReadInt64();
int ReadInt32();
short ReadInt16();
```

The preceding example is preferable to the following language-specific alternative.

VB

```
ReadDouble() As Double
ReadSingle() As Single
ReadLong() As Long
ReadInteger() As Integer
ReadShort() As Short
[C#]
double ReadDouble();
float ReadFloat();
long ReadLong();
int ReadInt();
short ReadShort();
```

See Also

[Design Guidelines for Class Library Developers](#) | [Common Type System](#)

Namespace Naming Guidelines

.NET Framework 1.1

The general rule for naming namespaces is to use the company name followed by the technology name and optionally the feature and design as follows.

```
CompanyName. TechnologyName[. Feature][. Design]
```

For example:

```
Microsoft. Media  
Microsoft. Media. Design
```

Prefixing namespace names with a company name or other well-established brand avoids the possibility of two published namespaces having the same name. For example, `Microsoft.Office` is an appropriate prefix for the Office Automation Classes provided by Microsoft.

Use a stable, recognized technology name at the second level of a hierarchical name. Use organizational hierarchies as the basis for namespace hierarchies. Name a namespace that contains types that provide design-time functionality for a base namespace with the `.Design` suffix. For example, the [System.Windows.Forms.Design Namespace](#) contains designers and related classes used to design [System.Windows.Forms](#) based applications.

A nested namespace should have a dependency on types in the containing namespace. For example, the classes in the [System.Web.UI.Design](#) depend on the classes in [System.Web.UI](#). However, the classes in **System.Web.UI** do not depend on the classes in **System.Web.UI.Design**.

You should use [Pascal case](#) for namespaces, and separate logical components with periods, as in `Microsoft.Office.PowerPoint`. If your brand employs nontraditional casing, follow the casing defined by your brand, even if it deviates from the prescribed Pascal case. For example, the namespaces `Next.WebObjects` and `ee.cummings` illustrate appropriate deviations from the Pascal case rule.

Use plural namespace names if it is semantically appropriate. For example, use `System.Collections` rather than `System.Collection`. Exceptions to this rule are brand names and abbreviations. For example, use `System.IO` rather than `System.IOs`.

Do not use the same name for a namespace and a class. For example, do not provide both a `Debug` namespace and a `Debug` class.

Finally, note that a namespace name does not have to parallel an assembly name. For example, if you name an assembly `MyCompany.MyTechnology.dll`, it does not have to contain a `MyCompany.MyTechnology` namespace.

See Also

[Design Guidelines for Class Library Developers](#) | [Introduction to the .NET Framework Class Library](#)

Class Naming Guidelines

.NET Framework 1.1

The following rules outline the guidelines for naming classes:

- Use a noun or noun phrase to name a class.
- Use [Pascal case](#).
- Use abbreviations sparingly.
- Do not use a type prefix, such as `C` for class, on a class name. For example, use the class name `FileStream` rather than `CFileStream`.
- Do not use the underscore character (`_`).
- Occasionally, it is necessary to provide a class name that begins with the letter `I`, even though the class is not an interface. This is appropriate as long as `I` is the first letter of an entire word that is a part of the class name. For example, the class name `IdentityStore` is appropriate.
- Where appropriate, use a compound word to name a derived class. The second part of the derived class's name should be the name of the base class. For example, `ApplicationException` is an appropriate name for a class derived from a class named `Exception`, because `ApplicationException` is a kind of `Exception`. Use reasonable judgment in applying this rule. For example, `Button` is an appropriate name for a class derived from `Control`. Although a button is a kind of control, making `Control` a part of the class name would lengthen the name unnecessarily.

The following are examples of correctly named classes.

VB

```
Public Class FileStream
Public Class Button
Public Class String
[C#]
public class FileStream
public class Button
public class String
```

See Also

[Design Guidelines for Class Library Developers](#) | [Base Class Usage Guidelines](#)

Interface Naming Guidelines

.NET Framework 1.1

The following rules outline the naming guidelines for interfaces:

- Name interfaces with nouns or noun phrases, or adjectives that describe behavior. For example, the interface name **IComponent** uses a descriptive noun. The interface name **ICustomAttributeProvider** uses a noun phrase. The name **IPersistable** uses an adjective.
- Use [Pascal case](#).
- Use abbreviations sparingly.
- Prefix interface names with the letter **I**, to indicate that the type is an interface.
- Use similar names when you define a class/interface pair where the class is a standard implementation of the interface. The names should differ only by the letter **I** prefix on the interface name.
- Do not use the underscore character (**_**).

The following are examples of correctly named interfaces.

VB

```
Public Interface IServiceProvider
Public Interface IFormatable
[C#]
public interface IServiceProvider
public interface IFormatable
```

The following code example illustrates how to define the interface **IComponent** and its standard implementation, the class **Component**.

VB

```
Public Interface IComponent
' Implementation code goes here.
End Interface

Public Class Component
Implements IComponent
' Implementation code goes here.
End Class

[C#]
public interface IComponent
{
// Implementation code goes here.
}
public class Component: IComponent
{
// Implementation code goes here.
}
```

See Also

[Design Guidelines for Class Library Developers](#) | [Base Class Usage Guidelines](#)

Attribute Naming Guidelines

.NET Framework 1.1

You should always add the suffix `Attribute` to custom attribute classes. The following is an example of a correctly named attribute class.

VB

```
Public Class ObsoleteAttribute  
[C#]  
public class ObsoleteAttribute{}
```

See Also

[Design Guidelines for Class Library Developers](#)

Enumeration Type Naming Guidelines

.NET Framework 1.1

The enumeration (**Enum**) value type inherits from the [Enum Class](#). The following rules outline the naming guidelines for enumerations:

- Use [Pascal case](#) for **Enum** types and value names.
- Use abbreviations sparingly.
- Do not use an `Enum` suffix on **Enum** type names.
- Use a singular name for most **Enum** types, but use a plural name for **Enum** types that are bit fields.
- Always add the **FlagsAttribute** to a bit field **Enum** type.

See Also

[Design Guidelines for Class Library Developers](#) | [Value Type Usage Guidelines](#) | [Enum Class](#)

Static Field Naming Guidelines

.NET Framework 1.1

The following rules outline the naming guidelines for static fields:

- Use nouns, noun phrases, or abbreviations of nouns to name static fields.
- Use [Pascal case](#).
- Do not use a Hungarian notation prefix on static field names.
- It is recommended that you use static properties instead of public static fields whenever possible.

See Also

[Design Guidelines for Class Library Developers](#) | [Field Usage Guidelines](#)

Parameter Naming Guidelines

.NET Framework 1.1

It is important to carefully follow these parameter naming guidelines because visual design tools that provide context sensitive help and class browsing functionality display method parameter names to users in the designer. The following rules outline the naming guidelines for parameters:

- Use [camel case](#) for parameter names.
- Use descriptive parameter names. Parameter names should be descriptive enough that the name of the parameter and its type can be used to determine its meaning in most scenarios. For example, visual design tools that provide context sensitive help display method parameters to the developer as they type. The parameter names should be descriptive enough in this scenario to allow the developer to supply the correct parameters.
- Use names that describe a parameter's meaning rather than names that describe a parameter's type. Development tools should provide meaningful information about a parameter's type. Therefore, a parameter's name can be put to better use by describing meaning. Use type-based parameter names sparingly and only where it is appropriate.
- Do not use reserved parameters. Reserved parameters are private parameters that might be exposed in a future version if they are needed. Instead, if more data is needed in a future version of your class library, add a new overload for a method.
- Do not prefix parameter names with Hungarian type notation.

The following are examples of correctly named parameters.

VB

```
GetType(typeName As String)As Type
Format(format As String, args() As object)As String
[C#]
Type GetType(string typeName)
string Format(string format, object[] args)
```

See Also

[Design Guidelines for Class Library Developers | Parameter Usage Guidelines](#)

Method Naming Guidelines

.NET Framework 1.1

The following rules outline the naming guidelines for methods:

- Use verbs or verb phrases to name methods.
- Use [Pascal case](#).

The following are examples of correctly named methods.

```
RemoveAll ()  
GetCharArray()  
Invoke()
```

See Also

[Design Guidelines for Class Library Developers](#) | [Method Usage Guidelines](#)

Property Naming Guidelines

.NET Framework 1.1

The following rules outline the naming guidelines for properties:

- Use a noun or noun phrase to name properties.
- Use [Pascal case](#).
- Do not use Hungarian notation.
- Consider creating a property with the same name as its underlying type. For example, if you declare a property named Color, the type of the property should likewise be [Color](#). See the example later in this topic.

The following code example illustrates correct property naming.

VB

```
Public Class SampleClass
    Public Property BackColor As Color
        ' Code for Get and Set accessors goes here.
    End Property
End Class
[C#]
public class SampleClass
{
    public Color BackColor
    {
        // Code for Get and Set accessors goes here.
    }
}
```

The following code example illustrates providing a property with the same name as a type.

VB

```
Public Enum Color
    ' Insert code for Enum here.
End Enum
Public Class Control
    Public Property Color As Color
        Get
            ' Insert code here.
        End Get
        Set
            ' Insert code here.
        End Set
    End Property
End Class
[C#]
public enum Color
{
    // Insert code for Enum here.
}
public class Control
{
    public Color Color
    {
        get { // Insert code here. }
        set { // Insert code here. }
    }
}
```

The following code example is incorrect because the property Color is of type Integer.

VB

```
Public Enum Color
    ' Insert code for Enum here.
End Enum
Public Class Control
    Public Property Color As Integer
        Get
            ' Insert code here.
        End Get
        Set
            ' Insert code here.
        End Set
    End Property
End Class
[C#]
public enum Color { // Insert code for Enum here. }
public class Control
{
    public int Color
```

```
{
    get { // Insert code here. }
    set { // Insert code here. }
}
```

In the incorrect example, it is not possible to refer to the members of the `Color` enumeration. `Color.xxx` will be interpreted as accessing a member that first gets the value of the **Color** property (type **Integer** in Visual Basic or type **int** in C#) and then accesses a member of that value (which would have to be an instance member of **System.Int32**).

See Also

[Design Guidelines for Class Library Developers](#) | [Property Usage Guidelines](#)

Event Naming Guidelines

.NET Framework 1.1

The following rules outline the naming guidelines for events:

- Use [Pascal case](#).
- Do not use Hungarian notation.
- Use an `EventHandler` suffix on event handler names.
- Specify two parameters named *sender* and *e*. The *sender* parameter represents the object that raised the event. The *sender* parameter is always of type **object**, even if it is possible to use a more specific type. The state associated with the event is encapsulated in an instance of an event class named *e*. Use an appropriate and specific event class for the *e* parameter type.
- Name an event argument class with the `EventArgs` suffix.
- Consider naming events with a verb. For example, correctly named event names include **Clicked**, **Painting**, and **DroppedDown**.
- Use a gerund (the "ing" form of a verb) to create an event name that expresses the concept of pre-event, and a past-tense verb to represent post-event. For example, a **Close** event that can be canceled should have a `Closing` event and a `Closed` event. Do not use the `BeforeXxx/AfterXxx` naming pattern.
- Do not use a prefix or suffix on the event declaration on the type. For example, use `Close` instead of `OnClose`.
- In general, you should provide a protected method called `OnXxx` on types with events that can be overridden in a derived class. This method should only have the event parameter *e*, because the sender is always the instance of the type.

The following example illustrates an event handler with an appropriate name and parameters.

VB

```
Public Delegate Sub MouseEventHandler(sender As Object, e As MouseEventArgs)
[C#]
public delegate void MouseEventHandler(object sender, MouseEventArgs e);
```

The following example illustrates a correctly named event argument class.

VB

```
Public Class MouseEventArgs
    Inherits EventArgs
    Dim x As Integer
    Dim y As Integer

    Public Sub New MouseEventArgs(x As Integer, y As Integer)
        me.x = x
        me.y = y
    End Sub

    Public Property X As Integer
        Get
            Return x
        End Get
    End Property

    Public Property Y As Integer
        Get
            Return y
        End Get
    End Property
End Class
[C#]
public class MouseEventArgs : EventArgs
{
    int x;
    int y;
    public MouseEventArgs(int x, int y)
    { this.x = x; this.y = y; }
    public int X { get { return x; } }
    public int Y { get { return y; } }
}
```

See Also

[Design Guidelines for Class Library Developers | Event Usage Guidelines](#)

Class Member Usage Guidelines

.NET Framework 1.1

This topic provides guidelines for using class members in class libraries.

In This Section

[Property Usage Guidelines](#)

Describes the guidelines to follow when using properties in class libraries.

[Event Usage Guidelines](#)

Describes the guidelines to follow when using events in class libraries.

[Method Usage Guidelines](#)

Describes the guidelines to follow when using and overloading methods in class libraries.

[Constructor Usage Guidelines](#)

Describes the guidelines to follow when using constructors in class libraries.

[Field Usage Guidelines](#)

Describes the guidelines to follow when using fields in class libraries.

[Parameter Usage Guidelines](#)

Describes the guidelines to follow when using parameters in class libraries.

Related Sections

[Property Naming Guidelines](#)

Describes the guidelines to follow when naming properties.

[Event Naming Guidelines](#)

Describes the guidelines to follow when naming events.

[Method Naming Guidelines](#)

Describes the guidelines to follow when naming methods.

[Static Field Naming Guidelines](#)

Describes the guidelines to follow when naming static fields.

[Parameter Naming Guidelines](#)

Describes the guidelines to follow when naming parameters.

[Design Guidelines for Class Library Developers](#)

Provides naming and usage guidelines for types in the .NET Framework as well as guidelines for implementing common design patterns.

Property Usage Guidelines

.NET Framework 1.1

Determine whether a property or a method is more appropriate for your needs. For details on choosing between properties and methods, see [Properties vs. Methods](#).

Choose a name for your property based on the recommended [Property Naming Guidelines](#).

When accessing a property using the **set** accessor, preserve the value of the property before you change it. This will ensure that data is not lost if the **set** accessor throws an exception.

Property State Issues

Allow properties to be set in any order. Properties should be stateless with respect to other properties. It is often the case that a particular feature of an object will not take effect until the developer specifies a particular set of properties, or until an object has a particular state. Until the object is in the correct state, the feature is not active. When the object is in the correct state, the feature automatically activates itself without requiring an explicit call. The semantics are the same regardless of the order in which the developer sets the property values or how the developer gets the object into the active state.

For example, a **TextBox** control might have two related properties: **DataSource** and **DataField**. **DataSource** specifies the table name, and **DataField** specifies the column name. Once both properties are specified, the control can automatically bind data from the table into the **Text** property of the control. The following code example illustrates properties that can be set in any order.

VB

```
Dim t As New TextBox()  
t.DataSource = "Publishers"  
t.DataField = "AuthorID"  
' The data-binding feature is now active.  
[C#]  
TextBox t = new TextBox();  
t.DataSource = "Publishers";  
t.DataField = "AuthorID";  
// The data-binding feature is now active.
```

You can set the **DataSource** and **DataField** properties in any order. Therefore, the preceding code is equivalent to the following.

VB

```
Dim t As New TextBox()  
t.DataField = "AuthorID"  
t.DataSource = "Publishers"  
' The data-binding feature is now active.  
[C#]  
TextBox t = new TextBox();  
t.DataField = "AuthorID";  
t.DataSource = "Publishers";  
// The data-binding feature is now active.
```

You can also set a property to null (**Nothing** in Visual Basic) to indicate that the value is not specified.

VB

```
Dim t As New TextBox()  
t.DataField = "AuthorID"  
t.DataSource = "Publishers"  
' The data-binding feature is now active.  
t.DataSource = Nothing  
' The data-binding feature is now inactive.  
[C#]  
TextBox t = new TextBox();  
t.DataField = "AuthorID";  
t.DataSource = "Publishers";  
// The data-binding feature is now active.  
t.DataSource = null;  
// The data-binding feature is now inactive.
```

The following code example illustrates how to track the state of the data binding feature and automatically activate or deactivate it at the appropriate times.

VB

```
Public Class TextBox  
    Private m_dataSource As String  
    Private m_dataField As String  
    Private m_active As Boolean  
  
    Public Property DataSource() As String  
        Get  
            Return m_dataSource  
        End Get  
        Set  
            If value <> m_dataSource Then  
                ' Set the property value first, in case activate fails.
```

```

        m_dataSource = value
        ' Update active state.
        SetActive(( Not (m_dataSource Is Nothing) And Not (m_dataField Is Nothing)))
    End If
End Set
End Property
Public Property DataField() As String
    Get
        Return m_dataField
    End Get
    Set
        If value <> m_dataField Then
            ' Set the property value first, in case activate fails.
            m_dataField = value
            ' Update active state.
            SetActive(( Not (m_dataSource Is Nothing) And Not (m_dataField Is Nothing)))
        End If
    End Set
End Property
Sub SetActive(m_value As Boolean)
    If value <> m_active Then
        If m_value Then
            Activate()
            Text = dataBase.Value(m_dataField)
        Else
            Deactivate()
            Text = ""
        End If
        ' Set active only if successful.
        m_active = value
    End If
End Sub
Sub Activate()
    ' Open database.
End Sub

Sub Deactivate()
    ' Close database.
End Sub
End Class
[C#]
public class TextBox
{
    string dataSource;
    string dataField;
    bool active;

    public string DataSource
    {
        get
        {
            return dataSource;
        }
        set
        {
            if (value != dataSource)
            {
                // Update active state.
                SetActive(value != null && dataField != null);
                dataSource = value;
            }
        }
    }

    public string DataField
    {
        get
        {
            return dataField;
        }
        set
        {
            if (value != dataField)
            {
                // Update active state.
                SetActive(dataSource != null && dataField != null);
                dataField = value;
            }
        }
    }
}
void SetActive(Boolean value)
{
    if (value != active)
    {
        if (value)
        {
            Activate();
            Text = dataBase.Value(dataField);

```

```

    }
    else
    {
        Deactivate();
        Text = "";
    }
    // Set active only if successful.
    active = value;
}
}
void Activate()
{
    // Open database.
}

void Deactivate()
{
    // Close database.
}
}

```

In the preceding example, the following expression determines whether the object is in a state in which the data-binding feature can activate itself.

VB

```

(Not (value Is Nothing) And Not (m_dataField Is Nothing))
[C#]
value != null && dataField != null

```

You make activation automatic by creating a method that determines whether the object can be activated given its current state, and then activates it as necessary.

VB

```

Sub UpdateActive(m_dataSource As String, m_dataField As String)
    SetActive(( Not (m_dataSource Is Nothing) And Not (m_dataField Is Nothing)))
End Sub
[C#]
void UpdateActive(string dataSource, string dataField)
{
    SetActive(dataSource != null && dataField != null);
}

```

If you do have related properties, such as **DataSource** and **DataMember**, you should consider implementing the [ISupportInitialize Interface](#). This will allow the designer (or user) to call the [ISupportInitialize.BeginInit](#) and [ISupportInitialize.EndInit](#) methods when setting multiple properties to allow the component to provide optimizations. In the above example, **ISupportInitialize** could prevent unnecessary attempts to access the database until setup is correctly completed.

The expression that appears in this method indicates the parts of the object model that need to be examined in order to enforce these state transitions. In this case, the **DataSource** and **DataField** properties are affected. For more information on choosing between properties and methods, see [Properties vs. Methods](#).

Raising Property-Changed Events

Components should raise property-changed events if they want to notify consumers when the component's property changes programmatically. The naming convention for a property-changed event is to add the **Changed** suffix to the property name, such as **TextChanged**. For example, a control might raise a **TextChanged** event when its text property changes. You can use a protected helper routine **Raise<Property>Changed**, to raise this event. However, it is probably not worth the overhead to raise a property-changed event for a hash table item addition. The following code example illustrates the implementation of a helper routine on a property-changed event.

VB

```

Class Control
    Inherits Component
    Private m_text As String
    Public Property Text() As String
        Get
            Return m_text
        End Get
        Set
            If Not m_text.Equals(value) Then
                m_text = value
                RaiseTextChanged()
            End If
        End Set
    End Property
End Class
[C#]
class Control: Component
{
    string text;
    public string Text
    {
        get
        {
            return text;
        }
        set
        {
            if (!text.Equals(value))

```

```

        {
            text = value;
            RaiseTextChanged();
        }
    }
}

```

Data binding uses this pattern to allow two-way binding of the property. Without **<Property>Changed** and **Raise<Property>Changed** events, data binding works in one direction; if the database changes, the property is updated. Each property that raises the **<Property>Changed** event should provide metadata to indicate that the property supports data binding.

It is recommended that you raise changing/changed events if the value of a property changes as a result of external forces. These events indicate to the developer that the value of a property is changing or has changed as a result of an operation, rather than by calling methods on the object.

A good example is the **Text** property of an **Edit** control. As a user types information into the control, the property value automatically changes. An event is raised before the value of the property has changed. It does not pass the old or new value, and the developer can cancel the event by throwing an exception. The name of the event is the name of the property followed by the suffix **Changing**. The following code example illustrates a changing event.

VB

```

Class Edit
    Inherits Control

    Public Property Text() As String
        Get
            Return m_text
        End Get
        Set
            If m_text <> value Then
                OnTextChanged(Event.Empty)
                m_text = value
            End If
        End Set
    End Property
End Class
[C#]
class Edit : Control
{
    public string Text
    {
        get
        {
            return text;
        }
        set
        {
            if (text != value)
            {
                OnTextChanged(Event.Empty);
                text = value;
            }
        }
    }
}

```

An event is also raised after the value of the property has changed. This event cannot be canceled. The name of the event is the name of the property followed by the suffix **Changed**. The generic **PropertyChanged** event should also be raised. The pattern for raising both of these events is to raise the specific event from the **OnPropertyChanged** method. The following example illustrates the use of the **OnPropertyChanged** method.

VB

```

Class Edit
    Inherits Control
    Public Property Text() As String
        Get
            Return m_text
        End Get
        Set
            If m_text <> value Then
                OnTextChanged(Event.Empty)
                m_text = value
                RaisePropertyChangedEvent(Edit.ClassInfo, m_text)
            End If
        End Set
    End Property
    Protected Sub OnPropertyChanged(e As PropertyChangedEventArgs)
        If e.PropertyChanged.Equals(Edit.ClassInfo, m_text) Then
            OnTextChanged(Event.Empty)
        End If
        If Not (onPropertyChangedHandler Is Nothing) Then
            onPropertyChangedHandler(Me, e)
        End If
    End Sub
End Class
[C#]

```

```

class Edit : Control
{
    public string Text
    {
        get
        {
            return text;
        }
        set
        {
            if (text != value)
            {
                OnTextChanged(Event.Empty);
                text = value;
                RaisePropertyChangedEvent(Edit.ClassInfo, text);
            }
        }
    }

    protected void OnPropertyChanged(PropertyChangedEventArgs e)
    {
        if (e.PropertyChanged.Equals(Edit.ClassInfo, text))
            OnTextChanged(Event.Empty);
        if (onPropertyChangedHandler != null)
            onPropertyChangedHandler(this, e);
    }
}

```

There are cases when the underlying value of a property is not stored as a field, making it difficult to track changes to the value. When raising the changing event, find all the places that the property value can change and provide the ability to cancel the event. For example, the previous **Edit** control example is not entirely accurate because the **Text** value is actually stored in the window handle (**HWND**). In order to raise the **TextChanged** event, you must examine Windows messages to determine when the text might change, and allow for an exception thrown in **OnTextChanged** to cancel the event. If it is too difficult to provide a changing event, it is reasonable to support only the changed event.

Properties vs. Methods

Class library designers often must decide between implementing a class member as a property or a method. In general, methods represent actions and properties represent data. Use the following guidelines to help you choose between these options.

- Use a property when the member is a logical data member. In the following member declarations, **Name** is a property because it is a logical member of the class.

VB

```

Public Property Name As String
    Get
        Return m_name
    End Get
    Set
        m_name = value
    End Set
End Property
[C#]
public string Name
{
    get
    {
        return name;
    }
    set
    {
        name = value;
    }
}

```

- Use a method when:
 - The operation is a conversion, such as **Object.ToString**.
 - The operation is expensive enough that you want to communicate to the user that they should consider caching the result.
 - Obtaining a property value using the **get** accessor would have an observable side effect.
 - Calling the member twice in succession produces different results.
 - The order of execution is important. Note that a type's properties should be able to be set and retrieved in any order.
 - The member is static but returns a value that can be changed.
 - The member returns an array. Properties that return arrays can be very misleading. Usually it is necessary to return a copy of the internal array so that the user cannot change internal state. This, coupled with the fact that a user can easily assume it is an indexed property, leads to inefficient code. In the following code example, each call to the **Methods** property creates a copy of the array. As a result, $2^n + 1$ copies of the array will be created in the following loop.

VB

```

Dim type As Type = ' Get a type.
Dim i As Integer
For i = 0 To type.Methods.Length - 1
    If type.Methods(i).Name.Equals("text") Then
        ' Perform some operation.
    End If
Next i
[C#]
Type type = // Get a type.
for (int i = 0; i < type.Methods.Length; i++)
{
    if (type.Methods[i].Name.Equals("text"))
    {

```

```

        }
        // Perform some operation.
    }
}

```

The following example illustrates the correct use of properties and methods.

VB

```

Class Connection
    ' The following three members should be properties
    ' because they can be set in any order.
    Property DNSName() As String
        ' Code for get and set accessors goes here.
    End Property
    Property UserName() As String
        ' Code for get and set accessors goes here.
    End Property
    Property Password() As String
        ' Code for get and set accessors goes here.
    End Property
    ' The following member should be a method
    ' because the order of execution is important.
    ' This method cannot be executed until after the
    ' properties have been set.
    Function Execute() As Boolean

```

```

[C#]
class Connection
{
    // The following three members should be properties
    // because they can be set in any order.
    string DNSName {get();set();}
    string UserName {get();set();}
    string Password {get();set();}

    // The following member should be a method
    // because the order of execution is important.
    // This method cannot be executed until after the
    // properties have been set.
    bool Execute ();
}

```

Read-Only and Write-Only Properties

You should use a read-only property when the user cannot change the property's logical data member. Do not use write-only properties.

Indexed Property Usage

Note An indexed property can also be referred to as an indexer.

The following rules outline guidelines for using indexed properties:

- Use an indexed property when the property's logical data member is an array.
- Consider using only integral values or strings for an indexed property. If the design requires other types for the indexed property, reconsider whether it represents a logical data member. If not, use a method.
- Consider using only one index. If the design requires multiple indexes, reconsider whether it represents a logical data member. If not, use a method.
- Use only one indexed property per class, and make it the default indexed property for that class. This rule is enforced by indexer support in the C# programming language.
- Do not use nondefault indexed properties. C# does not allow this.
- Name an indexed property `Item`. For example, see the [DataGrid.Item Property](#). Follow this rule, unless there is a name that is more obvious to users, such as the `chars` property on the **String** class. In C#, indexers are always named `Item`.
- Do not provide an indexed property and a method that are semantically equivalent to two or more overloaded methods. In the following code example, the `Method` property should be changed to `GetMethod(string)` method. Note that this not allowed in C#.

VB

```

' Change the MethodInfo Type.Method property to a method.
Property Type.Method(name As String) As MethodInfo
Function Type.GetMethod(name As String, ignoreCase As Boolean) As MethodInfo
[C#]
// Change the MethodInfo Type.Method property to a method.
MethodInfo Type.Method[string name]
MethodInfo Type.GetMethod (string name, Boolean ignoreCase)
[Visual Basic]
' The MethodInfo Type.Method property is changed to
' the MethodInfo Type.GetMethod method.
Function Type.GetMethod(name As String) As MethodInfo
Function Type.GetMethod(name As String, ignoreCase As Boolean) As MethodInfo
[C#]
// The MethodInfo Type.Method property is changed to
// the MethodInfo Type.GetMethod method.
MethodInfo Type.GetMethod(string name)
MethodInfo Type.GetMethod (string name, Boolean ignoreCase)

```

See Also

[Design Guidelines for Class Library Developers](#) | [Property Naming Guidelines](#) | [Class Member Usage Guidelines](#)

Event Usage Guidelines

.NET Framework 1.1

The following rules outline the usage guidelines for events:

- Choose a name for your event based on the recommended [Event Naming Guidelines](#).
- When you refer to events in documentation, use the phrase, "an event was raised" instead of "an event was fired" or "an event was triggered."
- In languages that support the **void** keyword, use a return type of void for event handlers, as shown in the following C# code example.

```
public delegate void MouseEventHandler(object sender, MouseEventArgs e);
```

- Use strongly typed event data classes when an event conveys meaningful data, such as the coordinates of a mouse click.
- Event classes should extend the [System.EventArgs Class](#), as shown in the following example.

VB

```
Public Class MouseEventArgs
    Inherits EventArgs
    ' Code for the class goes here.
End Class
[C#]
public class MouseEventArgs {}
```

- Use a **protected** (**Protected** in Visual Basic) virtual method to raise each event. This technique is not appropriate for sealed classes, because classes cannot be derived from them. The purpose of the method is to provide a way for a derived class to handle the event using an override. This is more natural than using delegates in situations where the developer is creating a derived class. The name of the method takes the form OnEventName, where EventName is the name of the event being raised. For example:

VB

```
Public Class Button
    Private onClickHandler As ButtonClickHandler
    Protected Overridable Sub OnClick(e As ClickEventArgs)
        ' Call the delegate if non-null.
        If Not (onClickHandler Is Nothing) Then
            onClickHandler(Me, e)
        End If
    End Sub
End Class

[C#]
public class Button
{
    ButtonClickHandler onClickHandler;

    protected virtual void OnClick(ClickEventArgs e)
    {
        // Call the delegate if non-null.
        if (onClickHandler != null)
            onClickHandler(this, e);
    }
}
```

The derived class can choose not to call the base class during the processing of OnEventName. Be prepared for this by not including any processing in the OnEventName method that is required for the base class to work correctly.

- You should assume that an event handler could contain any code. Classes should be ready for the event handler to perform almost any operation, and in all cases the object should be left in an appropriate state after the event has been raised. Consider using a try/finally block at the point in code where the event is raised. Since the developer can perform a callback function on the object to perform other actions, do not assume anything about the object state when control returns to the point at which the event was raised. For example:

VB

```
Public Class Button
    Private onClickHandler As ButtonClickHandler
    Protected Sub DoClick()
        ' Paint button in indented state.
        PaintDown()
        Try
            ' Call event handler.
            OnClick()
        Finally
            ' Window might be deleted in event handler.
            If Not (windowHandle Is Nothing) Then
                ' Paint button in normal state.
                PaintUp()
            End If
        End Try
    End Sub
    Protected Overridable Sub OnClick(e As ClickEvent)
        If Not (onClickHandler Is Nothing) Then
            onClickHandler(Me, e)
        End If
    End Sub
End Class
```



```

    End Sub
End Class
[C#]
public class Button
{
    ButtonClickHandler onClickHandler;

    protected void DoClick()
    {
        // Paint button in indented state.
        PaintDown();
        try
        {
            // Call event handler.
            OnClick();
        }
        finally
        {
            // Window might be deleted in event handler.
            if (windowHandler != null)
                // Paint button in normal state.
                PaintUp();
        }
    }

    protected virtual void OnClick(ClickEventArgs e)
    {
        if (onClickHandler != null)
            onClickHandler(this, e);
    }
}

```

- Use or extend the [System.ComponentModel.CancelEventArgs Class](#) to allow the developer to control the events of an object. For example, the [TreeView](#) control raises a **BeforeLabelEdit** when the user is about to edit a node label. The following code example illustrates how a developer can use this event to prevent a node from being edited.

VB

```

Public Class Form1
    Inherits Form
    Private treeView1 As New TreeView()

    Sub treeView1_BeforeLabelEdit(source As Object, e As NodeLabelEditEventArgs)
        e.CancelEdit = True
    End Sub
End Class
[C#]
public class Form1: Form
{
    TreeView treeView1 = new TreeView();

    void treeView1_BeforeLabelEdit(object source,
        NodeLabelEditEventArgs e)
    {
        e.CancelEdit = true;
    }
}

```

Note that in this case, no error is generated to the user. The label is read-only.

Cancel events are not appropriate in cases where the developer would cancel the operation and return an exception. In these cases, you should raise an exception inside of the event handler in order to cancel. For example, the user might want to write validation logic in an edit control as shown.

VB

```

Public Class Form1
    Inherits Form
    Private edit1 As EditText = New EditText()

    Sub TextChanging(source As Object, e As EventArgs)
        Throw New RuntimeException("Invalid edit")
    End Sub
End Class
[C#]
public class Form1: Form
{
    EditText edit1 = new EditText();

    void TextChanging(object source, EventArgs e)
    {
        throw new RuntimeException("Invalid edit");
    }
}

```

See Also

[Design Guidelines for Class Library Developers](#) | [Event Naming Guidelines](#) | [Class Member Usage Guidelines](#)

Method Usage Guidelines

.NET Framework 1.1

The following rules outline the usage guidelines for methods:

- Choose a name for your event based on the recommended [Method Naming Guidelines](#).
- Do not use Hungarian notation.
- By default, methods are nonvirtual. Maintain this default in situations where it is not necessary to provide virtual methods. For more information about implementing inheritance, see [Base Class Usage Guidelines](#).

Method Overloading Guidelines

Method overloading occurs when a class contains two methods with the same name, but different signatures. This section provides some guidelines for the use of overloaded methods.

- Use method overloading to provide different methods that do semantically the same thing.
- Use method overloading instead of allowing default arguments. Default arguments do not version well and therefore are not allowed in the [Common Language Specification](#) (CLS). The following code example illustrates an overloaded `String.IndexOf` method.

VB

```
Function String.IndexOf(name As String) As Integer
Function String.IndexOf(name As String, startIndex As Integer) As Integer
[C#]
int String.IndexOf (String name);
int String.IndexOf (String name, int startIndex);
```

- Use default values correctly. In a family of overloaded methods, the complex method should use parameter names that indicate a change from the default state assumed in the simple method. For example, in the following code, the first method assumes the search will not be case-sensitive. The second method uses the name `ignoreCase` rather than `caseSensitive` to indicate how the default behavior is being changed.

VB

```
' Method #1: ignoreCase = false.
Function Type.GetMethod(name As String) As MethodInfo
' Method #2: Indicates how the default behavior of method #1
' is being changed.
Function Type.GetMethod(name As String, ignoreCase As Boolean) As MethodInfo
[C#]
// Method #1: ignoreCase = false.
MethodInfo Type.GetMethod(String name);
// Method #2: Indicates how the default behavior of method #1 is being // changed.
MethodInfo Type.GetMethod (String name, Boolean ignoreCase);
```

- Use a consistent ordering and naming pattern for method parameters. It is common to provide a set of overloaded methods with an increasing number of parameters to allow the developer to specify a desired level of information. The more parameters that you specify, the more detail the developer can specify. In the following code example, the overloaded `Execute` method has a consistent parameter order and naming pattern variation. Each of the `Execute` method variations uses the same semantics for the shared set of parameters.

VB

```
Public Class SampleClass
    Private defaultForA As String = "default value for a"
    Private defaultForB As Integer = "42"
    Private defaultForC As Double = "68.90"

    Overloads Public Sub Execute()
        Execute(defaultForA, defaultForB, defaultForC)
    End Sub

    Overloads Public Sub Execute(a As String)
        Execute(a, defaultForB, defaultForC)
    End Sub

    Overloads Public Sub Execute(a As String, b As Integer)
        Execute(a, b, defaultForC)
    End Sub

    Overloads Public Sub Execute(a As String, b As Integer, c As Double)
        Console.WriteLine(a)
        Console.WriteLine(b)
        Console.WriteLine(c)
        Console.WriteLine()
    End Sub
End Class
[C#]
public class SampleClass
{
    readonly string defaultForA = "default value for a";
    readonly int defaultForB = "42";
    readonly double defaultForC = "68.90";

    public void Execute()
    {
        Execute(defaultForA, defaultForB, defaultForC);
    }
}
```

```

    }

    public void Execute (string a)
    {
        Execute(a, default tForB, default tForC);
    }

    public void Execute (string a, int b)
    {
        Execute (a, b, default tForC);
    }

    public void Execute (string a, int b, double c)
    {
        Console.WriteLine(a);
        Console.WriteLine(b);
        Console.WriteLine(c);
        Console.WriteLine();
    }
}

```

Note that the only method in the group that should be virtual is the one that has the most parameters and only when you need extensibility.

- If you must provide the ability to override a method, make only the most complete overload virtual and define the other operations in terms of it. The following example illustrates this pattern.

VB

```

Public Class SampleClass
    Private myString As String

    Public Sub New(str As String)
        Me.myString = str
    End Sub

    Overloads Public Function IndexOf(s As String) As Integer
        Return IndexOf(s, 0)
    End Function

    Overloads Public Function IndexOf(s As String, startIndex As Integer) As Integer
        Return IndexOf(s, startIndex, myString.Length - startIndex)
    End Function

    Overloads Public Overridable Function IndexOf(s As String,
        startIndex As Integer, count As Integer) As Integer
        Return myString.IndexOf(s, startIndex, count)
    End Function
End Class
[C#]
public class SampleClass
{
    private string myString;

    public MyClass(string str)
    {
        this.myString = str;
    }

    public int IndexOf(string s)
    {
        return IndexOf (s, 0);
    }

    public int IndexOf(string s, int startIndex)
    {
        return IndexOf(s, startIndex, myString.Length - startIndex );
    }

    public virtual int IndexOf(string s, int startIndex, int count)
    {
        return myString.IndexOf(s, startIndex, count);
    }
}

```

Methods With Variable Numbers of Arguments

You might want to expose a method that takes a variable number of arguments. A classic example is the **printf** method in the C programming language. For managed class libraries, use the **params** (**ParamArray** in Visual Basic) keyword for this construct. For example, use the following code instead of several overloaded methods.

VB

```

Sub Format(formatString As String, ParamArray args() As Object)
[C#]
void Format(string formatString, params object [] args)

```

You should not use the **VarArgs** or ellipsis (...) calling convention exclusively because the [Common Language Specification](#) does not support it.

For extremely performance-sensitive code, you might want to provide special code paths for a small number of elements. You should only do this if you are going to special case the entire code path (not just create an array and call the more general method). In such cases, the following pattern is recommended as a balance between performance and the cost of specially cased code.

VB

```
Sub Format(formatString As String, arg1 As Object)
Sub Format(formatString As String, arg1 As Object, arg2 As Object)

Sub Format(formatString As String, ParamArray args() As Object)
[C#]
void Format(string formatString, object arg1)
void Format(string formatString, object arg1, object arg2)

void Format(string formatString, params object [] args)
```

See Also

[Design Guidelines for Class Library Developers](#) | [Method Naming Guidelines](#) | [Class Member Usage Guidelines](#) | [Base Class Usage Guidelines](#)

Constructor Usage Guidelines

.NET Framework 1.1

The following rules outline the usage guidelines for constructors:

- Provide a default private constructor if there are only static methods and properties on a class. In the following example, the private constructor prevents the class from being created.

VB

```
NotInheritable Public Class Environment
    ' Private constructor prevents the class from being created.
    Private Sub New()
        ' Code for the constructor goes here.
    End Sub
End Class
[C#]
public sealed class Environment
{
    // Private constructor prevents the class from being created.
    private Environment()
    {
        // Code for the constructor goes here.
    }
}
```

- Minimize the amount of work done in the constructor. Constructors should not do more than capture the constructor parameter or parameters. This delays the cost of performing further operations until the user uses a specific feature of the instance.
- Provide a constructor for every class. If a type is not meant to be created, use a private constructor. If you do not specify a constructor, many programming language (such as C#) implicitly add a default public constructor. If the class is abstract, it adds a protected constructor. Be aware that if you add a nondefault constructor to a class in a later version release, the implicit default constructor will be removed which can break client code. Therefore, the best practice is to always explicitly specify the constructor even if it is a public default constructor.
- Provide a **protected** (**Protected** in Visual Basic) constructor that can be used by types in a derived class.
- You should not provide constructor without parameters for a value type **struct**. Note that many compilers do not allow a **struct** to have a constructor without parameters. If you do not supply a constructor, the runtime initializes all the fields of the **struct** to zero. This makes array and static field creation faster.
- Use parameters in constructors as shortcuts for setting properties. There should be no difference in semantics between using an empty constructor followed by property **set** accessors, and using a constructor with multiple arguments. The following three code examples are equivalent:

VB

```
' Example #1.
Dim SampleClass As New Class()
SampleClass.A = "a"
SampleClass.B = "b"

' Example #2.
Dim SampleClass As New Class("a")
SampleClass.B = "b"

' Example #3.
Dim SampleClass As New Class("a", "b")
[C#]
// Example #1.
Class SampleClass = new Class();
SampleClass.A = "a";
SampleClass.B = "b";

// Example #2.
Class SampleClass = new Class("a");
SampleClass.B = "b";

// Example #3.
Class SampleClass = new Class ("a", "b");
```

- Use a consistent ordering and naming pattern for constructor parameters. A common pattern for constructor parameters is to provide an increasing number of parameters to allow the developer to specify a desired level of information. The more parameters that you specify, the more detail the developer can specify. In the following code example, there is a consistent order and naming of the parameters for all the `SampleClass` constructors.

VB

```
Public Class SampleClass
    Private Const defaultForA As String = "default value for a"
    Private Const defaultForB As String = "default value for b"
    Private Const defaultForC As String = "default value for c"

    Public Sub New()
        MyClass.New(defaultForA, defaultForB, defaultForC)
        Console.WriteLine("New()")
    End Sub

    Public Sub New(a As String)
        MyClass.New(a, defaultForB, defaultForC)
    End Sub
```

```

Public Sub New(a As String, b As String)
    MyClass.New(a, b, default tForC)
End Sub
Public Sub New(a As String, b As String, c As String)
    Me.a = a
    Me.b = b
    Me.c = c
End Sub
End Class
[C#]
public class SampleClass
{
    private const string default tForA = "default t value for a";
    private const string default tForB = "default t value for b";
    private const string default tForC = "default t value for c";

    public MyClass(): this(default tForA, default tForB, default tForC) {}
    public MyClass (string a) : this(a, default tForB, default tForC) {}
    public MyClass (string a, string b) : this(a, b, default tForC) {}
    public MyClass (string a, string b, string c)
}

```

See Also

[Design Guidelines for Class Library Developers](#) | [Class Member Usage Guidelines](#)

Field Usage Guidelines

.NET Framework 1.1

The following rules outline the usage guidelines for fields:

- Do not use instance fields that are **public** or **protected** (**Public** or **Protected** in Visual Basic). If you avoid exposing fields directly to the developer, classes can be versioned more easily because a field cannot be changed to a property while maintaining binary compatibility. Consider providing **get** and **set** property accessors for fields instead of making them public. The presence of executable code in **get** and **set** property accessors allows later improvements, such as creation of an object on demand, upon usage of the property, or upon a property change notification. The following code example illustrates the correct use of private instance fields with **get** and **set** property accessors.

VB

```
Public Structure Point
    Private xValue As Integer
    Private yValue As Integer

    Public Sub New(x As Integer, y As Integer)
        Me.xValue = x
        Me.yValue = y
    End Sub

    Public Property X() As Integer
        Get
            Return xValue
        End Get
        Set
            xValue = value
        End Set
    End Property
    Public Property Y() As Integer
        Get
            Return yValue
        End Get
        Set
            yValue = value
        End Set
    End Property
End Structure
[C#]
public struct Point
{
    private int xValue;
    private int yValue;

    public Point(int x, int y)
    {
        this.xValue = x;
        this.yValue = y;
    }

    public int X
    {
        get
        {
            return xValue;
        }
        set
        {
            xValue = value;
        }
    }

    public int Y
    {
        get
        {
            return yValue;
        }
        set
        {
            yValue = value;
        }
    }
}
```

- Expose a field to a derived class by using a **protected** property that returns the value of the field. This is illustrated in the following code example.

VB

```
Public Class Control
    Inherits Component
    Private handle As Integer

    Protected ReadOnly Property Handle() As Integer
        Get
```



```

        Return handle
    End Get
End Property
End Class
[C#]
public class Control : Component
{
    private int handle;
    protected int Handle
    {
        get
        {
            return handle;
        }
    }
}

```

- Use the **const** (**Const** in Visual Basic) keyword to declare constant fields that will not change. Language compilers save the values of **const** fields directly in calling code.
- Use public static read-only fields for predefined object instances. If there are predefined instances of an object, declare them as public static read-only fields of the object itself. Use **Pascal case** because the fields are public. The following code example illustrates the correct use of public static read-only fields.

VB

```

Public Structure Color
    Public Shared Red As New Color(&HFF)
    Public Shared Green As New Color(&HFF00)
    Public Shared Blue As New Color(&HFF0000)
    Public Shared Black As New Color(&H0)
    Public Shared White As New Color(&FFFFFF)

    Public Sub New(rgb As Integer)
        ' Insert code here.
    End Sub

    Public Sub New(r As Byte, g As Byte, b As Byte)
        ' Insert code here.
    End Sub

    Public ReadOnly Property RedValue() As Byte
        Get
            Return Color.Red
        End Get
    End Property

    Public ReadOnly Property GreenValue() As Byte
        Get
            Return Color.Green
        End Get
    End Property

    Public ReadOnly Property BlueValue() As Byte
        Get
            Return Color.Blue
        End Get
    End Property
End Structure
[C#]
public struct Color
{
    public static readonly Color Red = new Color(0x0000FF);
    public static readonly Color Green = new Color(0x00FF00);
    public static readonly Color Blue = new Color(0xFF0000);
    public static readonly Color Black = new Color(0x000000);
    public static readonly Color White = new Color(0xFFFFFFFF);

    public Color(int rgb)
    { // Insert code here. }
    public Color(byte r, byte g, byte b)
    { // Insert code here. }

    public byte RedValue
    {
        get
        {
            return Color.Red;
        }
    }
    public byte GreenValue
    {
        get
        {
            return Color.Green;
        }
    }
    public byte BlueValue
    {
        get

```

```
    {  
        return Color;  
    }  
}
```

- Spell out all words used in a field name. Use abbreviations only if developers generally understand them. Do not use uppercase letters for field names. The following is an example of correctly named fields.

VB

```
Class SampleClass  
    Private url As String  
    Private destinationUrl As String  
End Class  
[C#]  
class SampleClass  
{  
    string url;  
    string destinationUrl;  
}
```

- Do not use Hungarian notation for field names. Good names describe semantics, not type.
- Do not apply a prefix to field names or static field names. Specifically, do not apply a prefix to a field name to distinguish between static and nonstatic fields. For example, applying a `g_` or `s_` prefix is incorrect.

See Also

[Design Guidelines for Class Library Developers](#) | [Class Member Usage Guidelines](#) | [Static Field Naming Guidelines](#)

Parameter Usage Guidelines

.NET Framework 1.1

The following rules outline the usage guidelines for parameters:

- Check for valid parameter arguments. Perform argument validation for every public or protected method and property **set** accessor. Throw meaningful exceptions to the developer for invalid parameter arguments. Use the [System.ArgumentException Class](#), or a class derived from **System.ArgumentException**. The following example checks for valid parameter arguments and throws meaningful exceptions.

VB

```
Class SampleClass
    Private countValue As Integer
    Private maxValue As Integer = 100

    Public Property Count() As Integer
        Get
            Return countValue
        End Get
        Set
            ' Check for valid parameter.
            If value < 0 Or value >= maxValue Then
                Throw New ArgumentException("value", value,
                    "Value is invalid.")
            End If
            countValue = value
        End Set
    End Property
    Public Sub SelectItem(start As Integer, [end] As Integer)
        ' Check for valid parameter.
        If start < 0 Then
            Throw New ArgumentOutOfRangeException("start", start, "Start
                is invalid.")
        End If
        ' Check for valid parameter.
        If [end] < start Then
            Throw New ArgumentOutOfRangeException("end", [end], "End is
                invalid.")
        End If
        ' Insert code to do other work here.
        Console.WriteLine("Starting at {0}", start)
        Console.WriteLine("Ending at {0}", [end])
    End Sub
End Class
```

```
[C#]
class SampleClass
{
    public int Count
    {
        get
        {
            return count;
        }
        set
        {
            // Check for valid parameter.
            if (count < 0 || count >= MaxValue)
                throw new ArgumentOutOfRangeException(
                    Sys.GetStrin
                        "InvalidArgument", "value", count.ToString());
        }
    }

    public void Select(int start, int end)
    {
        // Check for valid parameter.
        if (start < 0)
            throw new ArgumentException(
                Sys.GetStrin("InvalidArgument", "start", start.ToString());
        // Check for valid parameter.
        if (end < start)
            throw new ArgumentException(
                Sys.GetStrin("InvalidArgument", "end", end.ToString());
    }
}
```

Note that the actual checking does not necessarily have to happen in the **public** or **protected** method itself. It could happen at a lower level in private routines. The main point is that the entire surface area that is exposed to the developer checks for valid arguments.

- Make sure you fully understand the implications of passing parameters by value or by reference. Passing a parameter by value copies the value being passed and has no effect on the original value. The following method example passes parameters by value.

```
public void Add(object value){}
```

Passing a parameter by reference passes the storage location for the value. As a result, changes can be made to the value of the parameter. The following method example passes a parameter by value.

```
public static int Exchange(ref int location, int value){}
```

An output parameter represents the same storage location as the variable specified as the argument in the method invocation. As a result, changes can be made only to the output parameter. The following method example passes an out parameter.

```
[DllImport("Kernel 32.dll")]  
public static extern bool QueryPerformanceCounter(out long value)
```

See Also

[Design Guidelines for Class Library Developers](#) | [Parameter Naming Guidelines](#) | [Class Member Usage Guidelines](#)

Type Usage Guidelines

.NET Framework 1.1

Types are the units of encapsulation in the common language runtime. For a detailed description of the complete list of data types supported by the runtime, see the [Common Type System](#). This section provides usage guidelines for the basic kinds of types.

In This Section

[Base Class Usage Guidelines](#)

Describes how to implement base classes and when to use interfaces instead of classes.

[Value Type Usage Guidelines](#)

Describes how to use the **Struct** and **Enum** value types.

[Delegate Usage Guidelines](#)

Describes how to use delegates for event notifications and callback functions.

[Attribute Usage Guidelines](#)

Describes how to use attributes as declarative information to describe types.

[Nested Type Usage Guidelines](#)

Describes how to use types defined within the scope of another type.

Related Sections

[Naming Guidelines](#)

Describes the guidelines for naming types in class libraries.

[Class Naming Guidelines](#)

Describes the guidelines to follow when naming classes.

[Attribute Naming Guidelines](#)

Describes the correct way to name an attribute using the `Attribute` suffix.

[Design Guidelines for Class Library Developers](#)

Provides naming and usage guidelines for types in the .NET Framework as well as guidelines for implementing common design patterns.

Base Class Usage Guidelines

.NET Framework 1.1

A class is the most common kind of type. A class can be abstract or sealed. An abstract class requires a derived class to provide an implementation. A sealed class does not allow a derived class. It is recommended that you use classes over other types.

Base classes are a useful way to group objects that share a common set of functionality. Base classes can provide a default set of functionality, while allowing customization through extension.

You should explicitly provide a constructor for a class. Compilers commonly add a public default constructor to classes that do not define a constructor. This can be misleading to a user of the class, if your intention is for the class not to be creatable. Therefore, it is best practice to always define at least one constructor for a class. If you do not want it to be creatable, make the constructor private.

You should add extensibility or polymorphism to your design only if you have a clear customer scenario for it. For example, providing an interface for data adapters is difficult and serves no real benefit. Developers will still have to program against each adapter specifically, so there is only marginal benefit from providing an interface. However, you do need to support consistency between all adapters. Although an interface or abstract class is not appropriate in this situation, providing a consistent pattern is very important. You can provide consistent patterns for developers in base classes. Follow these guidelines for creating base classes.

Base Classes vs. Interfaces

An interface type is a specification of a protocol, potentially supported by many object types. Use base classes instead of interfaces whenever possible. From a versioning perspective, classes are more flexible than interfaces. With a class, you can ship Version 1.0 and then in Version 2.0 add a new method to the class. As long as the method is not abstract, any existing derived classes continue to function unchanged.

Because interfaces do not support implementation inheritance, the pattern that applies to classes does not apply to interfaces. Adding a method to an interface is equivalent to adding an abstract method to a base class; any class that implements the interface will break because the class does not implement the new method.

Interfaces are appropriate in the following situations:

- Several unrelated classes want to support the protocol.
- These classes already have established base classes (for example, some are user interface (UI) controls, and some are XML Web services).
- Aggregation is not appropriate or practical.

In all other situations, class inheritance is a better model.

Protected Methods and Constructors

Provide class customization through protected methods. The public interface of a base class should provide a rich set of functionality for the consumer of the class. However, users of the class often want to implement the fewest number of methods possible to provide that rich set of functionality to the consumer. To meet this goal, provide a set of nonvirtual or final public methods that call through to a single protected method that provides implementations for the methods. This method should be marked with the `Implements` suffix. Using this pattern is also referred to as providing a Template method. The following code example demonstrates this process.

VB

```
Public Class SampleClass

    Private x As Integer
    Private y As Integer
    Private width As Integer
    Private height As Integer
    Private specified As BoundsSpecified

    Overloads Public Sub SetBounds(x As Integer, y As Integer, width As Integer, height As Integer)
        SetBoundsImpl(x, y, width, height, Me.specified)
    End Sub

    Overloads Public Sub SetBounds(x As Integer, y As Integer, width As Integer, height As Integer, specified As BoundsSpecified)
        SetBoundsImpl(x, y, width, height, specified)
    End Sub

    Protected Overridable Sub SetBoundsImpl(x As Integer, y As Integer, width As Integer, height As Integer, specified As BoundsSpecified)
        ' Insert code to perform meaningful operations here.
        Me.x = x
        Me.y = y
        Me.width = width
        Me.height = height
        Me.specified = specified
        Console.WriteLine("x {0}, y {1}, width {2}, height {3}, bounds {4}", Me.x, Me.y, Me.width, Me.height, Me.specified)
    End Sub
End Class

[C#]
public class MyClass
{
    private int x;
    private int y;
    private int width;
    private int height;
    BoundsSpecified specified;
}
```

```
public void SetBounds(int x, int y, int width, int height)
{
    SetBoundsImpl(x, y, width, height, this.specificied);
}

public void SetBounds(int x, int y, int width, int height,
    BoundsSpecified specified)
{
    SetBoundsImpl(x, y, width, height, specified);
}

protected virtual void SetBoundsImpl(int x, int y, int width, int
    height, BoundsSpecified specified)
{
    // Add code to perform meaningful operations here.
    this.x = x;
    this.y = y;
    this.width = width;
    this.height = height;
    this.specificied = specified;
}
}
```

Many compilers, such as the C# compiler, insert a **public** or **protected** constructor if you do not. Therefore, for better documentation and readability of your source code, you should explicitly define a **protected** constructor on all abstract classes.

Sealed Class Usage Guidelines

Use sealed classes if there are only static methods and properties on a class.

See Also

[Design Guidelines for Class Library Developers](#) | [Type Usage Guidelines](#) | [Class Naming Guidelines](#)

Value Type Usage Guidelines

.NET Framework 1.1

A value type describes a value that is represented as a sequence of bits stored on the stack. For a description of all the .NET Framework's built-in data types, see [Value Types](#). This section provides guidelines for using the structure (**struct**) and enumeration (**enum**) value types.

Struct Usage Guidelines

It is recommended that you use a **struct** for types that meet any of the following criteria:

- Act like primitive types.
- Have an instance size under 16 bytes.
- Are immutable.
- Value semantics are desirable.

The following example shows a correctly defined structure.

VB

```
Public Structure Int32
    Implements IFormattable
    Implements IComparable
    Public Const MinValue As Integer = -2147483648
    Public Const MaxValue As Integer = 2147483647

    Private intValue As Integer

    Overloads Public Shared Function ToString(i As Integer) As String
        ' Insert code here.
    End Function

    Overloads Public Function ToString(ByVal format As String, ByVal
        formatProvider As IFormatProvider) As String Implements
        IFormattable.ToString
        ' Insert code here.
    End Function

    Overloads Public Overrides Function ToString() As String
        ' Insert code here.
    End Function
    Public Shared Function Parse(s As String) As Integer
        ' Insert code here.
        Return 0
    End Function

    Public Overrides Function GetHashCode() As Integer
        ' Insert code here.
        Return 0
    End Function

    Public Overrides Overloads Function Equals(obj As Object) As Boolean
        ' Insert code here.
        Return False
    End Function

    Public Function CompareTo(obj As Object) As Integer Implements
        IComparable.CompareTo
        ' Insert code here.
        Return 0
    End Function
End Structure
```

```
[C#]
public struct Int32: IComparable, IFormattable
{
    public const int MinValue = -2147483648;
    public const int MaxValue = 2147483647;

    public static string ToString(int i)
    {
        // Insert code here.
    }

    public string ToString(string format, IFormatProvider formatProvider)
    {
        // Insert code here.
    }

    public override string ToString()
    {
        // Insert code here.
    }

    public static int Parse(string s)
    {
        // Insert code here.
    }
}
```



```

    return 0;
}

public override int GetHashCode()
{
    // Insert code here.
    return 0;
}

public override bool Equals(object obj)
{
    // Insert code here.
    return false;
}

public int CompareTo(object obj)
{
    // Insert code here.
    return 0;
}
}

```

- Do not provide a default constructor for a **struct**. Note that C# does not allow a **struct** to have a default constructor. The runtime inserts a constructor that initializes all the values to a zero state. This allows arrays of structs to be created without running the constructor on each instance. Do not make a **struct** dependent on a constructor being called for each instance. Instances of structs can be created with a zero value without running a constructor. You should also design a **struct** for a state where all instance data is set to zero, false, or null (as appropriate) to be valid.

Enum Usage Guidelines

The following rules outline the usage guidelines for enumerations:

- Do not use an `Enum` suffix on **enum** types.
- Use an **enum** to strongly type parameters, properties, and return types. Always define enumerated values using an **enum** if they are used in a parameter or property. This allows development tools to know the possible values for a property or parameter. The following example shows how to define an enum type.

```

VB
Public Enum FileMode
    Append
    Create
    CreateNew
    Open
    OpenOrCreate
    Truncate
End Enum
[C#]
public enum FileMode
{
    Append,
    Create,
    CreateNew,
    Open,
    OpenOrCreate,
    Truncate
}

```

The following example shows the constructor for a **FileStream** object that uses the **FileMode** enumeration.

```

VB
Public Sub New(ByVal path As String, ByVal mode As FileMode);
[C#]
public FileStream(string path, FileMode mode);

```

- Use an **enum** instead of static final constants.
- Do not use an **enum** for open sets (such as the operating system version).
- Use the `System.FlagsAttribute` Class to create custom attribute for an **enum** only if a bitwise OR operation is to be performed on the numeric values. Use powers of two for the **enum** values so that they can be easily combined. This attribute is applied in the following code example.

```

VB
<Flags(>
Public Enum WatcherChangeTypes
    Created = 1
    Deleted = 2
    Changed = 4
    Renamed = 8
    All = Created Or Deleted Or Changed Or Renamed
End Enum
[C#]
[Flags()]
public enum WatcherChangeTypes
{
    Created = 1,
    Deleted = 2,
    Changed = 4,
    Renamed = 8,
}

```

```
    All = Created | Deleted | Changed | Renamed
};
```

Note An exception to this rule is when encapsulating a Win32 API. It is common to have internal definitions that come from a Win32 header. You can leave these with the Win32 casing, which is usually all capital letters.

- Consider providing named constants for commonly used combinations of flags. Using the bitwise OR is an advanced concept and should not be required for simple tasks. This is illustrated in the following example of an enumeration.

VB

```
<Flags()> _
Public Enum FileAccess
    Read = 1
    Write = 2
    ReadWrite = Read Or Write
End Enum
[C#]
[Flags()]
public enum FileAccess
{
    Read = 1,
    Write = 2,
    ReadWrite = Read | Write,
}
```

- Use type **Int32** as the underlying type of an **enum** unless either of the following is true:
 - The **enum** represents flags and there are currently more than 32 flags, or the **enum** might grow to have many flags in the future.
 - The type needs to be different from **int** for backward compatibility.
- Do not assume that **enum** arguments will be in the defined range. It is valid to cast any integer value into an **enum** even if the value is not defined in the **enum**. Perform argument validation as illustrated in the following code example.

VB

```
Public Sub SetColor(newColor As Color)
    If Not [Enum].IsDefined(GetType(Color), newColor) Then
        Throw New ArgumentOutOfRangeException()
    End If
End Sub
[C#]
public void SetColor (Color color)
{
    if (!Enum.IsDefined (typeof(Color), color)
        throw new ArgumentOutOfRangeException();
}
```

See Also

[Design Guidelines for Class Library Developers](#) | [Enumeration Type Naming Guidelines](#) | [Value Types](#) | [Enumerations](#)

Delegate Usage Guidelines

.NET Framework 1.1

A delegate is a powerful tool that allows the managed code object model designer to encapsulate method calls. Delegates are useful for event notifications and callback functions.

Event notifications

Use the appropriate event design pattern for events even if the event is not user interface-related. For more information on using events, see the [Event Usage Guidelines](#).

Callback functions

Callback functions are passed to a method so that user code can be called multiple times during execution to provide customization. Passing a Compare callback function to a sort routine is a classic example of using a callback function. These methods should use the callback function conventions described in [Callback Function Usage](#).

Name end callback functions with the suffix `Callback`.

See Also

[Design Guidelines for Class Library Developers](#) | [Callback Function Usage](#)

Attribute Usage Guidelines

.NET Framework 1.1

The .NET Framework enables developers to invent new kinds of declarative information, to specify declarative information for various program entities, and to retrieve attribute information in a run-time environment. For example, a framework might define a `HelpAttribute` attribute that can be placed on program elements such as classes and methods to provide a mapping from program elements to their documentation. New kinds of declarative information are defined through the declaration of attribute classes, which might have positional and named parameters. For more information about attributes, see [Writing Custom Attributes](#).

The following rules outline the usage guidelines for attribute classes:

- Add the `Attribute` suffix to custom attribute classes, as shown in the following example.

VB

```
Public Class ObsoleteAttribute()  
[C#]  
public class ObsoleteAttribute()  

```

- Specify `AttributeUsage` on your attributes to define their usage precisely, as shown in the following example.

VB

```
<AttributeUsage(AttributeTargets.All, Inherited := False, AllowMultiple := True)> _  
  
Public Class ObsoleteAttribute  
    Inherits Attribute  
    ' Insert code here.  
End Class  
[C#]  
[AttributeUsage(AttributeTargets.All, Inherited = false, AllowMultiple = true)]  
public class ObsoleteAttribute: Attribute {  

```

- Seal attribute classes whenever possible, so that classes cannot be derived from them.
- Use positional arguments (constructor parameters) for required parameters. Provide a read-only property with the same name as each positional argument, but change the case to differentiate between them. This allows access to the argument at runtime.
- Use named arguments (read/write properties) for optional parameters. Provide a read/write property with the same name as each named argument, but change the case to differentiate between them.
- Do not define a parameter with both named and positional arguments. The following example illustrates this pattern.

VB

```
Public Class NameAttribute  
    Inherits Attribute  
  
    ' This is a positional argument.  
    Public Sub New(username As String)  
        ' Implement code here.  
    End Sub  
  
    Public ReadOnly Property UserName() As String  
        Get  
            Return UserName  
        End Get  
    End Property  
  
    ' This is a named argument.  
    Public Property Age() As Integer  
        Get  
            Return Age  
        End Get  
        Set  
            Age = value  
        End Set  
    End Property  
End Class  
[C#]  
public class NameAttribute: Attribute  
{  
    // This is a positional argument.  
    public NameAttribute (string username)  
    {  
        // Implement code here.  
    }  
    public string UserName  
    {  
        get  
        {  
            return UserName;  
        }  
    }  
    // This is a named argument.  
    public int Age  
    {  
        get  

```

```
    {  
        return Age;  
    }  
    set  
    {  
        Age = value;  
    }  
}
```

See Also

[Design Guidelines for Class Library Developers](#) | [Attribute Naming Guidelines](#)

Nested Type Usage Guidelines

.NET Framework 1.1

A nested type is a type defined within the scope of another type. Nested types are very useful for encapsulating implementation details of a type, such as an enumerator over a collection, because they can have access to private state.

Public nested types should be used rarely. Use them only in situations where both of the following are true:

- The nested type (inner type) logically belongs to the containing type (outer type).

The following examples illustrates how to define types with and without nested types:

VB

```
' With nested types.
ListBox.SelectedObjectCollection
' Without nested types.
ListBoxSelectedObjectCollection

' With nested types.
RichTextBox.ScrollBars
' Without nested types.
RichTextBoxScrollBars
[C#]
// With nested types.
ListBox.SelectedObjectCollection
// Without nested types.
ListBoxSelectedObjectCollection

// With nested types.
RichTextBox.ScrollBars
// Without nested types.
RichTextBoxScrollBars
```

Do not use nested types if the following are true:

- The type must be instantiated by client code. If a type has a public constructor, it probably should not be nested. The rationale behind this guideline is that if a nested type can be instantiated, it indicates that the type has a place in the library on its own. You can create it, use it, and destroy it without using the outer type. Therefore, it should not be nested. An inner type should not be widely reused outside of the outer type without a relationship to the outer type.
- References to the type are commonly declared in client code.

See Also

[Design Guidelines for Class Library Developers](#)

Guidelines for Exposing Functionality to COM

.NET Framework 1.1

The common language runtime provides rich support for interoperating with COM components. A COM component can be used from within a managed type and a managed instance can be used by a COM component. This support is the key to moving unmanaged code to managed code one piece at a time; however, it does present some issues for class library designers. In order to fully expose a managed type to COM clients, the type must expose functionality in a way that is supported by COM and abides by the COM versioning contract.

Mark managed class libraries with the [ComVisibleAttribute](#) attribute to indicate whether COM clients can use the library directly or whether they must use a wrapper that shapes the functionality so that they can use it.

Types and interfaces that must be used directly by COM clients, such as to host in an unmanaged container, should be marked with the **ComVisible(true)** attribute. The transitive closure of all types referenced by exposed types should be explicitly marked as **ComVisible(true)**; if not, they will be exposed as **IUnknown**.

Note Members of a type can also be marked as **ComVisible(false)**; this reduces exposure to COM and therefore reduces the restrictions on what a managed type can use.

Types marked with the **ComVisible(true)** attribute cannot expose functionality exclusively in a way that is not usable from COM. Specifically, COM does not support static methods or parameterized constructors. Test the type's functionality from COM clients to ensure correct behavior. Make sure that you understand the registry impact for making all types cocreateable.

Marshal By Reference

Marshal-by-reference objects are [Remotable Objects](#). Object remoting applies to three kinds of types:

- Types whose instances are copied when they are marshaled across an AppDomain boundary (on the same computer or a different computer). These types must be marked with the **Serializable** attribute.
- Types for which the runtime creates a transparent proxy when they are marshaled across an AppDomain boundary (on the same computer or a different computer). These types must ultimately be derived from [System.MarshalByRefObject Class](#).
- Types that are not marshaled across AppDomains at all. This is the default.

Follow these guidelines when using marshal by reference:

- By default, instances should be marshal-by-value objects. This means that their types should be marked as **Serializable**.
- Component types should be marshal-by-reference objects. This should already be the case for most components, because the common base class, [System.Component Class](#), is a marshal-by-reference class.
- If the type encapsulates an operating system resource, it should be a marshal-by-reference object. If the type implements the [IDisposable Interface](#) it will very likely have to be marshaled by reference. [System.IO.Stream](#) derives from [MarshalByRefObject](#). Most streams, such as FileStreams and NetworkStreams, encapsulate external resources, so they should be marshal-by-reference objects.
- Instances that simply hold state should be marshal-by-value objects (such as a **DataSet**).
- Special types that cannot be called across an AppDomain (such as a holder of static utility methods) should not be marked as **Serializable**.

See Also

[Design Guidelines for Class Library Developers](#) | [System.MarshalByRefObject Class](#) | [Exposing .NET Framework Components to COM](#)

Error Raising and Handling Guidelines

.NET Framework 1.1

The following rules outline the guidelines for raising and handling errors:

- All code paths that result in an exception should provide a method to check for success without throwing an exception. For example, to avoid a **FileNotFoundException** you can call **File.Exists**. This might not always be possible, but the goal is that under normal execution no exceptions should be thrown.
- End **Exception** class names with the **Exception** suffix as in the following code example.

VB

```
Public Class FileNotFoundException
    Inherits Exception
    ' Implementation code goes here.
End Class
[C#]
public class FileNotFoundException : Exception
{
    // Implementation code goes here.
}
```

- Use the common constructors shown in the following code example when creating exception classes.

VB

```
Public Class XxxException
    Inherits ApplicationException

    Public Sub New()
        ' Implementation code goes here.
    End Sub

    Public Sub New(message As String)
        ' Implementation code goes here.
    End Sub

    Public Sub New(message As String, inner As Exception)
        ' Implementation code goes here.
    End Sub

    Public Sub New(info As SerializationInfo, context As StreamingContext)
        ' Implementation code goes here.
    End Sub
End Class
[C#]
public class XxxException : ApplicationException
{
    public XxxException() { ... }
    public XxxException(string message) { ... }
    public XxxException(string message, Exception inner) { ... }
    public XxxException(SerializationInfo info, StreamingContext context) { ... }
}
```

- In most cases, use the predefined exception types. Only define new exception types for programmatic scenarios, where you expect users of your class library to catch exceptions of this new type and perform a programmatic action based on the exception type itself. This is in lieu of parsing the exception string, which would negatively impact performance and maintenance.
For example, it makes sense to define a **FileNotFoundException** because the developer might decide to create the missing file. However, a **FileIOException** is not something that would typically be handled specifically in code.
- Do not derive all new exceptions directly from the base class **SystemException**. Inherit from **SystemException** only when creating new exceptions in System namespaces. Inherit from **ApplicationException** when creating new exceptions in other namespaces.
- Group new exceptions derived from **SystemException** or **ApplicationException** by namespace. For example, all **System.IO** exceptions are grouped under **IOException** (derived from **SystemException**) and all Microsoft.Media exceptions could be grouped under **MediaException** (derived from **ApplicationException**).
- Use a localized description string in every exception. When the user sees an error message, it will be derived from the description string of the exception that was thrown, and never from the exception class.
- Create grammatically correct error messages with punctuation. Each sentence in the description string of an exception should end in a period. Code that generically displays an exception message to the user does not have to handle the case where a developer forgot the final period.
- Provide exception properties for programmatic access. Include extra information (other than the description string) in an exception only when there is a programmatic scenario where that additional information is useful. You should rarely need to include additional information in an exception.
- Do not expose privileged information in exception messages. Information such as paths on the local file system is considered privileged information. Malicious code could use this information to gather private user information from the computer.
- Do not use exceptions for normal or expected errors, or for normal flow of control.
- You should return null for extremely common error cases. For example, a **File.Open** command returns a null reference if the file is not found, but throws an exception if the file is locked.
- Design classes so that in the normal course of use an exception will never be thrown. In the following code example, a **FileStream** class exposes another way of determining if the end of the file has been reached to avoid the exception that will be thrown if the developer reads past the end of the file.

VB

```
Class FileRead
    Sub Open()
        Dim stream As FileStream = File.Open("myfile.txt", FileMode.Open)
        Dim b As Byte

        ' ReadByte returns -1 at end of file.
    End Sub
End Class
```



```

        While b = stream.ReadByte() <> true
            ' Do something.
        End While
    End Sub
End Class
[C#]
class FileRead
{
    void Open()
    {
        FileStream stream = File.Open("myfile.txt", FileMode.Open);
        byte b;

        // ReadByte returns -1 at end of file.
        while ((b = stream.ReadByte()) != true)
        {
            // Do something.
        }
    }
}

```

- Throw the [InvalidOperationException](#) exception if a call to a property **set** accessor or method is not appropriate given the object's current state.
- Throw an [ArgumentException](#) or create an exception derived from this class if invalid parameters are passed or detected.
- Be aware that the stack trace starts at the point where an exception is thrown, not where it is created with the **new** operator. Consider this when deciding where to throw an exception.
- Use the exception builder methods. It is common for a class to throw the same exception from different places in its implementation. To avoid repetitive code, use helper methods that create the exception using the **new** operator and return it. The following code example shows how to implement a helper method.

C#

```

class File
{
    string fileName;
    public byte[] Read(int bytes)
    {
        if (!ReadFile(handle, bytes))
            throw new FileNotFoundException();
    }

    FileNotFoundException NewFileNotFoundException()
    {
        string description =
            // Build localized string, include fileName.
        return new FileNotFoundException(description);
    }
}

```

- Throw exceptions instead of returning an error code or HRESULT.
- Throw the most specific exception possible.
- Create meaningful message text for exceptions, targeted at the developer.
- Set all fields on the exception you use.
- Use Inner exceptions (chained exceptions). However, do not catch and re-throw exceptions unless you are adding additional information or changing the type of the exception.
- Do not create methods that throw [NullReferenceException](#) or [IndexOutOfRangeException](#).
- Perform argument checking on protected (Family) and internal (Assembly) members. Clearly state in the documentation if the protected method does not do argument checking. Unless otherwise stated, assume that argument checking is performed. There might, however, be performance gains in not performing argument checking.
- Clean up any side effects when throwing an exception. Callers should be able to assume that there are no side effects when an exception is thrown from a function. For example, if a **Hashtable.Insert** method throws an exception, the caller can assume that the specified item was not added to the **Hashtable**.

Standard Exception Types

The following table lists the standard exceptions provided by the runtime and the conditions for which you should create a derived class.

Exception type	Base type	Description	Example
Exception	Object	Base class for all exceptions.	None (use a derived class of this exception).
SystemException	Exception	Base class for all runtime-generated errors.	None (use a derived class of this exception).
IndexOutOfRangeException	SystemException	Thrown by the runtime only when an array is indexed improperly.	Indexing an array outside of its valid range: arr[arr.Length+1]
NullReferenceException	SystemException	Thrown by the runtime only when a null object is referenced.	object o = null; o.ToString();
InvalidOperationException	SystemException	Thrown by methods when in an invalid state.	Calling Enumerator.GetNext() after removing an Item from the underlying collection.
ArgumentException	SystemException	Base class for all argument	None (use a derived class of

		exceptions.	this exception).
ArgumentNullException	ArgumentException	Thrown by methods that do not allow an argument to be null.	String s = null; "Calculate".IndexOf(s);
ArgumentOutOfRangeException	ArgumentException	Thrown by methods that verify that arguments are in a given range.	String s = "string"; s.Chars[9];
ExternalException	SystemException	Base class for exceptions that occur or are targeted at environments outside of the runtime.	None (use a derived class of this exception).
COMException	ExternalException	Exception encapsulating COM Hresult information.	Used in COM interop.
SEHException	ExternalException	Exception encapsulating Win32 structured Exception Handling information.	Used in unmanaged code Interop.

Wrapping Exceptions

Errors that occur at the same layer as a component should throw an exception that is meaningful to target users. In the following code example, the error message is targeted at users of the **TextReader** class, attempting to read from a stream.

VB

```

Public Class TextReader
    Public Function ReadLine() As String
        Try
            ' Read a line from the stream.
        Catch e As Exception
            Throw New IOException("Could not read from stream", e)
        End Try
    End Function
End Class
[C#]
public class TextReader
{
    public string ReadLine()
    {
        try
        {
            // Read a line from the stream.
        }
        catch (Exception e)
        {
            throw new IOException ("Could not read from stream", e);
        }
    }
}

```

See Also

[Design Guidelines for Class Library Developers | Handling and Throwing Exceptions](#)

Array Usage Guidelines

.NET Framework 1.1

For a general description of arrays and array usage see [Arrays](#), and [System.Array Class](#).

Arrays vs. Collections

Class library designers might need to make difficult decisions about when to use an array and when to return a collection. Although these types have similar usage models, they have different performance characteristics. In general, you should use a collection when **Add**, **Remove**, or other methods for manipulating the collection are supported.

For more information on using collections, see [Grouping Data in Collections](#).

Array Usage

Do not return an internal instance of an array. This allows calling code to change the array. The following example demonstrates how the array `badChars` can be changed by any code that accesses the `Path` property even though the property does not implement the set accessor.

VB

```
Imports System
Imports System.Collections
Imports Microsoft.VisualBasic

Public Class ExampleClass
    NotInherited Public Class Path
        Private Sub New()
        End Sub

        Private Property Path
            Get
            End Get
            Set
            End Set
        End Property

        Private Shared badChars() As Char = {Chr(34), "<", ">"c}

        Public Shared Function GetInvalidPathChars() As Char()
            Return badChars
        End Function

    End Class

    Public Shared Sub Main()
        ' The following code displays the elements of the
        ' array as expected.
        Dim c As Char
        For Each c In Path.GetInvalidPathChars()
            Console.WriteLine(c)
        Next c
        Console.WriteLine()

        ' The following code sets all the values to A.
        Path.GetInvalidPathChars()(0) = "A"c
        Path.GetInvalidPathChars()(1) = "A"c
        Path.GetInvalidPathChars()(2) = "A"c

        ' The following code displays the elements of the array to the
        ' console. Note that the values have changed.
        For Each c In Path.GetInvalidPathChars()
            Console.WriteLine(c)
        Next c
    End Sub
End Class
```

[C#]

```
using System;
using System.Collections;

public class ExampleClass
{
    public sealed class Path
    {
        private Path(){}
        private static char[] badChars = {'\"', '<', '>'};
        public static char[] GetInvalidPathChars()
        {
            return badChars;
        }
    }

    public static void Main()
    {
        // The following code displays the elements of the
        // array as expected.
        foreach(char c in Path.GetInvalidPathChars())
```

```

    {
        Console.WriteLine(c);
    }
    Console.WriteLine();

    // The following code sets all the values to A.
    Path.GetInvalidPathChars()[0] = 'A';
    Path.GetInvalidPathChars()[1] = 'A';
    Path.GetInvalidPathChars()[2] = 'A';

    // The following code displays the elements of the array to the
    // console. Note that the values have changed.
    foreach(char c in Path.GetInvalidPathChars())
    {
        Console.WriteLine(c);
    }
}

```

You can correct the problem in the preceding example by making the `badChars` collection **readonly** (**ReadOnly** in Visual Basic). Alternately, you can clone the `badChars` collection before returning. The following example demonstrates how to modify the `GetInvalidPathChars` method to return a clone of the `badChars` collection.

VB

```

Public Shared Function GetInvalidPathChars() As Char()
    Return CType(badChars.Clone(), Char())
End Function
[C#]
public static char[] GetInvalidPathChars()
{
    return (char[])badChars.Clone();
}

```

Do not use **readonly** (**ReadOnly** in Visual Basic) fields of arrays. If you do, the array is readonly and cannot be changed, but the elements in the array can be changed. The following example demonstrates how the elements of the readonly array `InvalidPathChars` can be changed.

C#

```

public sealed class Path
{
    private Path(){}
    public static readonly char[] InvalidPathChars = {'\\', '<', '>', '|'}
}
//The following code can be used to change the values in the array.
Path.InvalidPathChars[0] = 'A';

```

Using Indexed Properties in Collections

You should use an indexed property only as a default member of a collection class or interface. Do not create families of functions in noncollection types. A pattern of methods, such as **Add**, **Item**, and **Count**, signal that the type should be a collection.

Array Valued Properties

You should use collections to avoid code inefficiencies. In the following code example, each call to the `myObj` property creates a copy of the array. As a result, $2^n + 1$ copies of the array will be created in the following loop.

VB

```

Dim i As Integer
For i = 0 To obj.myObj.Count - 1
    DoSomething(obj.myObj(i))
Next i
[C#]
for (int i = 0; i < obj.myObj.Count; i++)
    DoSomething(obj.myObj[i]);

```

For more information, see the [Properties vs. Methods](#) topic.

Returning Empty Arrays

String and **Array** properties should never return a null reference. Null can be difficult to understand in this context. For example, a user might assume that the following code will work.

VB

```

Public Sub DoSomething()
    Dim s As String = SomeOtherFunc()
    If s.Length > 0 Then
        ' Do something else.
    End If
End Sub
[C#]
public void DoSomething()
{
    string s = SomeOtherFunc();
    if (s.Length > 0)
    {

```

```
} // Do something else.  
}
```

The general rule is that null, empty string (""), and empty (0 item) arrays should be treated the same way. Return an empty array instead of a null reference.

See Also

[Design Guidelines for Class Library Developers](#) | [System.Array Class](#).

Operator Overloading Usage Guidelines

.NET Framework 1.1

The following rules outline the guidelines for operator overloading:

- Define operators on value types that are logical built-in language types, such as the [System.Decimal Structure](#).
- Provide operator-overloading methods only in the class in which the methods are defined. The C# compiler enforces this guideline.
- Use the names and signature conventions described in the [Common Language Specification \(CLS\)](#). The C# compiler does this for you automatically.
- Use operator overloading in cases where it is immediately obvious what the result of the operation will be. For example, it makes sense to be able to subtract one Time value from another Time value and get a TimeSpan. However, it is not appropriate to use the **or** operator to create the union of two database queries, or to use **shift** to write to a stream.
- Overload operators in a symmetric manner. For example, if you overload the equality operator (==), you should also overload the not equal operator(!=).
- Provide alternate signatures. Most languages do not support operator overloading. For this reason, it is a CLS requirement for all types that overload operators to include a secondary method with an appropriate domain-specific name that provides the equivalent functionality. It is a Common Language Specification (CLS) requirement to provide this secondary method. The following example is CLS-compliant.

C#

```
public struct DateTime
{
    public static TimeSpan operator -(DateTime t1, DateTime t2) { }
    public static TimeSpan Subtract(DateTime t1, DateTime t2) { }
}
```

The following table contains a list of operator symbols and the corresponding alternative methods and operator names.

C++ operator symbol	Name of alternative method	Name of operator
Not defined	ToXxx or FromXxx	op_Implicit
Not defined	ToXxx or FromXxx	op_Explicit
+ (binary)	Add	op_Addition
- (binary)	Subtract	op_Subtraction
* (binary)	Multiply	op_Multiply
/	Divide	op_Division
%	Mod	op_Modulus
^	Xor	op_ExclusiveOr
& (binary)	BitwiseAnd	op_BitwiseAnd
	BitwiseOr	op_BitwiseOr
&&	And	op_LogicalAnd
	Or	op_LogicalOr
=	Assign	op_Assign
<<	LeftShift	op_LeftShift
>>	RightShift	op_RightShift
Not defined	LeftShift	op_SignedRightShift
Not defined	RightShift	op_UnsignedRightShift
==	Equals	op_Equality
>	Compare	op_GreaterThan
<	Compare	op_LessThan
!=	Compare	op_Inequality
>=	Compare	op_GreaterThanOrEqual
<=	Compare	op_LessThanOrEqual
*=	Multiply	op_MultiplicationAssignment
-=	Subtract	op_SubtractionAssignment

^ =	Xor	op_ExclusiveOrAssignment
<< =	LeftShift	op_LeftShiftAssignment
% =	Mod	op_ModulusAssignment
+ =	Add	op_AdditionAssignment
& =	BitwiseAnd	op_BitwiseAndAssignment
 =	BitwiseOr	op_BitwiseOrAssignment
,	None assigned	op_Comma
/ =	Divide	op_DivisionAssignment
--	Decrement	op_Decrement
++	Increment	op_Increment
- (unary)	Negate	op_UnaryNegation
+ (unary)	Plus	op_UnaryPlus
~	OnesComplement	op_OnesComplement

See Also

[Design Guidelines for Class Library Developers](#)

Guidelines for Implementing Equals and the Equality Operator (==)

.NET Framework 1.1

The following rules outline the guidelines for implementing the **Equals** method and the equality operator (==):

- Implement the **GetHashCode** method whenever you implement the **Equals** method. This keeps **Equals** and **GetHashCode** synchronized.
- Override the **Equals** method whenever you implement ==, and make them do the same thing. This allows infrastructure code such as [Hashtable](#) and [ArrayList](#), which use the **Equals** method, to behave the same way as user code written using ==.
- Override the **Equals** method any time you implement the [IComparable](#) interface.
- You should consider implementing operator overloading for the equality (==), not equal (!=), less than (<), and greater than (>) operators when you implement **IComparable**.
- Do not throw exceptions from the **Equals** or **GetHashCode** methods or the equality operator (==).

For related information on the **Equals** method, see [Implementing the Equals Method](#).

Implementing the Equality Operator (==) on Value Types

In most programming languages there is no default implementation of the equality operator (==) for value types. Therefore, you should overload == any time equality is meaningful.

You should consider implementing the **Equals** method on value types because the default implementation on [System.ValueType](#) will not perform as well as your custom implementation.

Implement == any time you override the **Equals** method.

Implementing the Equality Operator (==) on Reference Types

Most languages do provide a default implementation of the equality operator (==) for reference types. Therefore, you should use care when implementing == on reference types. Most reference types, even those that implement the **Equals** method, should not override ==.

Override == if your type is a base type such as a Point, String, BigInteger, and so on. Any time you consider overloading the addition (+) and subtraction (-) operators, you also should consider overloading ==.

See Also

[Design Guidelines for Class Library Developers](#) | [Implementing the Equals Method](#) | [Object.Equals Method](#)

Guidelines for Casting Types

.NET Framework 1.1

The following rules outline the usage guidelines for casts:

- Do not allow implicit casts that will result in a loss of precision. For example, there should not be an implicit cast from **Double** to **Int32**, but there might be one from **Int32** to **Int64**.
- Do not throw exceptions from implicit casts because it is very difficult for the developer to understand what is happening.
- Provide casts that operate on an entire object. The value that is cast should represent the entire object, not a member of an object. For example, it is not appropriate for a **Button** to cast to a string by returning its caption.
- Do not generate a semantically different value. For example, it is appropriate to convert a **DateTime** or **TimeSpan** into an **Int32**. The **Int32** still represents the time or duration. It does not, however, make sense to convert a file name string such as "c:\mybitmap.gif" into a **Bitmap** object.
- Do not cast values from different domains. Casts operate within a particular domain of values. For example, numbers and strings are different domains. It makes sense that an **Int32** can cast to a **Double**. However, it does not make sense for an **Int32** to cast to a **String**, because they are in different domains.

See Also

[Design Guidelines for Class Library Developers](#)

Common Design Patterns

.NET Framework 1.1

This topic provides guidelines for implementing common design patterns in class libraries.

In This Section

[Implementing Finalize and Dispose to Clean Up Unmanaged Resources](#)

Describes the recommended design pattern to implement in class libraries to clean up unmanaged resources using the **Finalize** and **Dispose** methods.

[Implementing the Equals Method](#)

Describes the guidelines to follow to implement the **Equals** method in class libraries.

[Callback Function Usage](#)

Describes when to use delegates, events, and interfaces to provide callback functionality.

[Timeout Usage](#)

Describes the guidelines for using time-outs in base class libraries to specify the maximum time a caller is willing to wait for completion of a method call.

Related Sections

[Design Guidelines for Class Library Developers](#)

Implementing Finalize and Dispose to Clean Up Unmanaged Resources

.NET Framework 1.1

Class instances often encapsulate control over resources that are not managed by the runtime, such as window handles (HWND), database connections, and so on. Therefore, you should provide both an explicit and an implicit way to free those resources. Provide implicit control by implementing the protected [Finalize Method](#) on an object (destructor syntax in C# and the Managed Extensions for C++). The garbage collector calls this method at some point after there are no longer any valid references to the object.

In some cases, you might want to provide programmers using an object with the ability to explicitly release these external resources before the garbage collector frees the object. If an external resource is scarce or expensive, better performance can be achieved if the programmer explicitly releases resources when they are no longer being used. To provide explicit control, implement the [Dispose method](#) provided by the [IDisposable Interface](#). The consumer of the object should call this method when it is done using the object. **Dispose** can be called even if other references to the object are alive.

Note that even when you provide explicit control by way of **Dispose**, you should provide implicit cleanup using the **Finalize** method. **Finalize** provides a backup to prevent resources from permanently leaking if the programmer fails to call **Dispose**.

For more information about implementing **Finalize** and **Dispose** to clean up unmanaged resources, see [Programming for Garbage Collection](#). The following code example illustrates the basic design pattern for implementing Dispose.

VB

```
' Design pattern for a base class.
Public Class Base
    Implements IDisposable
    ' Implement IDisposable.
    Public Overloads Sub Dispose() Implements IDisposable.Dispose
        Dispose(True)
        GC.SuppressFinalize(Me)
    End Sub

    Protected Overloads Overridable Sub Dispose(disposing As Boolean)
        If disposing Then
            ' Free other state (managed objects).
        End If
        ' Free your own state (unmanaged objects).
        ' Set large fields to null.
    End Sub

    Protected Overrides Sub Finalize()
        ' Simply call Dispose(False).
        Dispose(False)
    End Sub
End Class

' Design pattern for a derived class.
Public Class Derived
    Inherits Base

    Protected Overloads Overrides Sub Dispose(disposing As Boolean)
        If disposing Then
            ' Release managed resources.
        End If
        ' Release unmanaged resources.
        ' Set large fields to null.
        ' Call Dispose on your base class.
        MyBase.Dispose(disposing)
    End Sub
    ' The derived class does not have a Finalize method
    ' or a Dispose method with parameters because it inherits
    ' them from the base class.
End Class

[C#]
// Design pattern for a base class.
public class Base: IDisposable
{
    //Implement IDisposable.
    public void Dispose()
    {
        Dispose(true);
        GC.SuppressFinalize(this);
    }

    protected virtual void Dispose(bool disposing)
    {
        if (disposing)
        {
            // Free other state (managed objects).
        }
        // Free your own state (unmanaged objects).
        // Set large fields to null.
    }
}
```

```

// Use C# destructor syntax for finalization code.
~Base()
{
    // Simply call Dispose(false).
    Dispose(false);
}

// Design pattern for a derived class.
public class Derived: Base
{
    protected override void Dispose(bool disposing)
    {
        if (disposing)
        {
            // Release managed resources.
        }
        // Release unmanaged resources.
        // Set large fields to null.
        // Call Dispose on your base class.
        base.Dispose(disposing);
    }
    // The derived class does not have a Finalize method
    // or a Dispose method with parameters because it inherits
    // them from the base class.
}

```

For a more detailed code example illustrating the design pattern for implementing **Finalize** and **Dispose**, see [Implementing a Dispose Method](#).

Customizing a Dispose Method Name

Occasionally a domain-specific name is more appropriate than **Dispose**. For example, a file encapsulation might want to use the method name **Close**. In this case, implement **Dispose** privately and create a public **Close** method that calls **Dispose**. The following code example illustrates this pattern. You can replace **Close** with a method name appropriate to your domain.

VB

```

' Do not make this method overridable.
' A derived class should not be allowed
' to override this method.
Public Sub Close()
    ' Call the Dispose method with no parameters.
    Dispose()
End Sub
[C#]
// Do not make this method virtual.
// A derived class should not be allowed
// to override this method.
public void Close()
{
    // Call the Dispose method with no parameters.
    Dispose();
}

```

Finalize

The following rules outline the usage guidelines for the **Finalize** method:

- Only implement **Finalize** on objects that require finalization. There are performance costs associated with **Finalize** methods.
- If you require a **Finalize** method, you should consider implementing **IDisposable** to allow users of your class to avoid the cost of invoking the **Finalize** method.
- Do not make the **Finalize** method more visible. It should be **protected**, not **public**.
- An object's **Finalize** method should free any external resources that the object owns. Moreover, a **Finalize** method should release only resources that are held onto by the object. The **Finalize** method should not reference any other objects.
- Do not directly call a **Finalize** method on an object other than the object's base class. This is not a valid operation in the C# programming language.
- Call the **base.Finalize** method from an object's **Finalize** method.

Note The base class's **Finalize** method is called automatically with the C# and the Managed Extensions for C++ destructor syntax.

Dispose

The following rules outline the usage guidelines for the **Dispose** method:

- Implement the dispose design pattern on a type that encapsulates resources that explicitly need to be freed. Users can free external resources by calling the public **Dispose** method.
- Implement the dispose design pattern on a base type that commonly has derived types that hold on to resources, even if the base type does not. If the base type has a close method, often this indicates the need to implement **Dispose**. In such cases, do not implement a **Finalize** method on the base type. **Finalize** should be implemented in any derived types that introduce resources that require cleanup.
- Free any disposable resources a type owns in its **Dispose** method.
- After **Dispose** has been called on an instance, prevent the **Finalize** method from running by calling the [GC.SuppressFinalize Method](#). The exception to this rule is the rare situation in which work must be done in **Finalize** that is not covered by **Dispose**.
- Call the base class's **Dispose** method if it implements **IDisposable**.
- Do not assume that **Dispose** will be called. Unmanaged resources owned by a type should also be released in a **Finalize** method in the event that **Dispose** is not called.
- Throw an **ObjectDisposedException** from instance methods on this type (other than **Dispose**) when resources are already disposed. This rule does not apply to the **Dispose** method because it should be callable multiple times without throwing an exception.
- Propagate the calls to **Dispose** through the hierarchy of base types. The **Dispose** method should free all resources held by this object and any object owned by this object. For example, you can create an object like a **TextReader** that holds onto a **Stream** and an **Encoding**, both of which are created by the **TextReader** without the user's knowledge. Furthermore, both the **Stream** and the **Encoding** can acquire external resources. When you call the **Dispose** method on the **TextReader**, it should in turn call **Dispose** on the **Stream** and the **Encoding**, causing them to release their external resources.

- You should consider not allowing an object to be usable after its **Dispose** method has been called. Recreating an object that has already been disposed is a difficult pattern to implement.
- Allow a **Dispose** method to be called more than once without throwing an exception. The method should do nothing after the first call.

See Also

[Design Guidelines for Class Library Developers](#) | [Programming for Garbage Collection](#) | [IDisposable.Dispose Method](#) | [Object.Finalize Method](#)

Implementing the Equals Method

.NET Framework 1.1

For related information on implementing the equality operator (`==`), see [Guidelines for Implementing Equals and the Equality Operator \(==\)](#).

- Override the **GetHashCode** method to allow a type to work correctly in a hash table.
- Do not throw an exception in the implementation of an **Equals** method. Instead, return **false** for a null argument.
- Follow the contract defined on the [Object.Equals Method](#) as follows:
 - `x.Equals(x)` returns **true**.
 - `x.Equals(y)` returns the same value as `y.Equals(x)`.
 - `(x.Equals(y) && y.Equals(z))` returns **true** if and only if `x.Equals(z)` returns **true**.
 - Successive invocations of `x.Equals(y)` return the same value as long as the objects referenced by `x` and `y` are not modified.
 - `x.Equals(null)` returns false.
- For some kinds of objects, it is desirable to have **Equals** test for value equality instead of referential equality. Such implementations of **Equals** return **true** if the two objects have the same value, even if they are not the same instance. The definition of what constitutes an object's value is up to the implementer of the type, but it is typically some or all of the data stored in the instance variables of the object. For example, the value of a string is based on the characters of the string; the **Equals** method of the **String** class returns **true** for any two instances of a string that contain exactly the same characters in the same order.
- When the **Equals** method of a base class provides value equality, an override of **Equals** in a derived class should call the inherited implementation of **Equals**.
- If you are programming in a language that supports operator overloading, and you choose to overload the equality operator (`==`) for a specified type, that type should override the **Equals** method. Such implementations of the **Equals** method should return the same results as the equality operator. Following this guideline will help ensure that class library code using **Equals** (such as [ArrayList](#) and [Hashtable](#)) works in a manner that is consistent with the way the equality operator is used by application code.
- If you are implementing a value type, you should consider overriding the **Equals** method to gain increased performance over the default implementation of the **Equals** method on [System.ValueType](#). If you override **Equals** and the language supports operator overloading, you should overload the equality operator for your value type.
- If you are implementing reference types, you should consider overriding the **Equals** method on a reference type if your type looks like a base type such as a `Point`, `String`, `BigInteger`, and so on. Most reference types should not overload the equality operator, even if they override **Equals**. However, if you are implementing a reference type that is intended to have value semantics, such as a complex number type, you should override the equality operator.
- If you implement the [IComparable Interface](#) on a given type, you should override **Equals** on that type.

Examples

Implementing the Equals method

The following code example contains two calls to the default implementation of the **Equals** method.

VB

```
Imports System
Class SampleClass
    Public Shared Sub Main()
        Dim obj1 As New System.Object()
        Dim obj2 As New System.Object()
        Console.WriteLine(obj1.Equals(obj2))
        obj1 = obj2
        Console.WriteLine(obj1.Equals(obj2))
    End Sub
End Class
[C#]
using System;
class SampleClass
{
    public static void Main()
    {
        Object obj1 = new Object();
        Object obj2 = new Object();
        Console.WriteLine(obj1.Equals(obj2));
        obj1 = obj2;
        Console.WriteLine(obj1.Equals(obj2));
    }
}
```

The output of the preceding code is the following.

```
False
True
```

Overriding the Equals method

The following code example shows a `Point` class that overrides the **Equals** method to provide value equality and a class `Point3D`, which is derived from `Point`. Because the `Point` class's override of **Equals** is the first in the inheritance chain to introduce value equality, the **Equals** method of the base class (which is inherited from **Object** and checks for referential equality) is not invoked. However, `Point3D.Equals` invokes `Point.Equals` because `Point` implements **Equals** in a manner that provides value equality.

VB

```
Imports System

Class Point
    Private x As Integer
```

```

    Private y As Integer

    Public Overrides Overloads Function Equals(obj As Object) As Boolean
        ' Check for null values and compare run-time types.
        If obj Is Nothing Or Not Me.GetType() Is obj.GetType() Then
            Return False
        End If
        Dim p As Point = CType(obj, Point)
        Return Me.x = p.x And Me.y = p.y
    End Function
End Class

Class Point3D
    Inherits Point
    Private z As Integer

    Public Overrides Overloads Function Equals(obj As Object) As Boolean
        Return MyBase.Equals(obj) And z = CType(obj, Point3D).z
    End Function

    Public Overrides Function GetHashCode() As Integer
        Return MyBase.GetHashCode() ^ z
    End Function
End Class
[C#]
using System;
class Point: object
{
    int x, y;
    public override bool Equals(Object obj)
    {
        // Check for null values and compare run-time types.
        if (obj == null || GetType() != obj.GetType())
            return false;
        Point p = (Point)obj;
        return (x == p.x) && (y == p.y);
    }
    public override int GetHashCode()
    {
        return x ^ y;
    }
}

class Point3D: Point
{
    int z;
    public override bool Equals(Object obj)
    {
        return base.Equals(obj) && z == ((Point3D)obj).z;
    }
    public override int GetHashCode()
    {
        return base.GetHashCode() ^ z;
    }
}

```

The `Point.Equals` method checks that the *obj* argument is non-null and that it references an instance of the same type as this object. If either of the checks fail, the method returns **false**. The **Equals** method uses the [Object.GetType Method](#) to determine whether the run-time types of the two objects are identical. Note that **typeof** (**TypeOf** in Visual Basic) is not used here because it returns the static type. If the method had used a check of the form `obj is Point` instead, the check would return **true** in cases where *obj* is an instance of a class derived from `Point`, even though *obj* and the current instance are not of the same run-time type. Having verified that both objects are of the same type, the method casts *obj* to type `Point` and returns the result of comparing the instance variables of the two objects.

In `Point3D.Equals`, the inherited **Equals** method is invoked before anything else is done. The inherited **Equals** method checks to see that *obj* is not null, that *obj* is an instance of the same class as this object, and that the inherited instance variables match. Only when the inherited **Equals** returns **true**, does the method compare the instance variables introduced in the derived class. Specifically, the cast to `Point3D` is not executed unless *obj* has been determined to be of type `Point3D` or a class derived from `Point3D`.

Using the Equals method to compare instance variables

In the previous example, the equality operator (`==`) is used to compare the individual instance variables. In some cases, it is appropriate to use the **Equals** method to compare instance variables in an **Equals** implementation, as shown in the following example.

VB

```

Imports System

Class Rectangle
    Private a, b As Point

    Public Overrides Overloads Function Equals(obj As [Object]) As Boolean
        If obj Is Nothing Or Not Me.GetType() Is obj.GetType() Then
            Return False
        End If
        Dim r As Rectangle = CType(obj, Rectangle)
        ' Use Equals to compare instance variables.
        Return Me.a.Equals(r.a) And Me.b.Equals(r.b)
    End Function
End Class

```

```

End Function

Public Overrides Function GetHashCode() As Integer
    Return a.GetHashCode() ^ b.GetHashCode()
End Function
End Class
[C#]
using System;
class Rectangle
{
    Point a, b;
    public override bool Equals(Object obj)
    {
        if (obj == null || GetType() != obj.GetType()) return false;
        Rectangle r = (Rectangle)obj;
        // Use Equals to compare instance variables.
        return a.Equals(r.a) && b.Equals(r.b);
    }
    public override int GetHashCode()
    {
        return a.GetHashCode() ^ b.GetHashCode();
    }
}

```

Overloading the equality operator (==) and the Equals method

In some programming languages, such as C#, operator overloading is supported. When a type overloads ==, it should also override the **Equals** method to provide the same functionality. This is typically accomplished by writing the **Equals** method in terms of the overloaded equality operator (==), as in the following example.

C#

```

public struct Complex
{
    double re, im;
    public override bool Equals(Object obj)
    {
        return obj is Complex && this == (Complex)obj;
    }
    public override int GetHashCode()
    {
        return re.GetHashCode() ^ im.GetHashCode();
    }
    public static bool operator ==(Complex x, Complex y)
    {
        return x.re == y.re && x.im == y.im;
    }
    public static bool operator !=(Complex x, Complex y)
    {
        return !(x == y);
    }
}

```

Because `Complex` is a C# **struct** (a value type), it is known that no classes will be derived from `Complex`. Therefore, the **Equals** method does not need to compare the **GetType** results for each object. Instead it uses the **is** operator to check the type of the *obj* parameter.

See Also

[Design Guidelines for Class Library Developers | Guidelines for Implementing Equals and the Equality Operator \(==\)](#)

Callback Function Usage

.NET Framework 1.1

[Delegates](#), [Interfaces](#) and [Events](#) allow you to provide callback functionality. Each type has its own specific usage characteristics that make it better suited to particular situations.

Events

Use an event if the following are true:

- A method signs up for the callback function up front, typically through separate **Add** and **Remove** methods.
- Typically, more than one object will want notification of the event.
- You want end users to be able to easily add a listener to the notification in the visual designer.

Delegates

Use a delegate if the following are true:

- You want a C language style function pointer.
- You want a single callback function.
- You want registration to occur in the call or at construction time, not through a separate **Add** method.

Interfaces

Use an interface if the callback function requires complex behavior.

See Also

[Design Guidelines for Class Library Developers](#)

Time-Out Usage

.NET Framework 1.1

Use time-outs to specify the maximum time a caller is willing to wait for completion of a method call.

A time-out might take the form of a parameter to the method call as follows.

VB

```
server.PerformOperation(timeout)
[C#]
server.PerformOperation(timeout);
```

Alternately, a time-out can be used as a property on the server class as follows.

VB

```
server.Timeout = timeout
server.PerformOperation()
[C#]
server.Timeout = timeout;
server.PerformOperation();
```

You should favor the first approach, because the association between the operation and the time-out is clearer. The property-based approach might be better if the server class is designed to be a component used with visual designers.

Historically, time-outs have been represented by integers. Integer time-outs can be hard to use because it is not obvious what the unit of the time-out is, and it is difficult to translate units of time into the commonly used millisecond.

A better approach is to use the [TimeSpan](#) structure as the time-out type. **TimeSpan** solves the problems with integer time-outs mentioned above. The following code example shows how to use a time-out of type **TimeSpan**.

VB

```
Public Class Server
    Public Sub PerformOperation(timeout As TimeSpan timeout)
        ' Insert code for the method here.
    End Sub
End Class

Public Class TestClass
    Dim server As New Server()
    server.PerformOperation(New TimeSpan(0, 15, 0))
End Class
[C#]
public class Server
{
    void PerformOperation(TimeSpan timeout)
    {
        // Insert code for the method here.
    }
}

public class TestClass
{
    public Server server = new Server();
    server.PerformOperation(new TimeSpan(0, 15, 0));
}
```

If the time-out is set to `TimeSpan(0)`, the method should throw an exception if the operation is not immediately completed. If the time-out is `TimeSpan.MaxValue`, the operation should wait forever without timing out, as if there were no time-out set. A server class is not required to support either of these values, but it should throw an **InvalidArgumentException** if an unsupported time-out value is specified.

If a time-out expires and an exception is thrown, the server class should cancel the underlying operation.

If a default time-out is used, the server class should include a static `defaultTimeout` property to be used if the user does not specify one. The following code example includes a static `OperationTimeout` property of type **TimeSpan** that returns `defaultTimeout`.

VB

```
Class Server
    Private defaultTimeout As New TimeSpan(1000)

    Overloads Sub PerformOperation()
        Me.PerformOperation(OperationTimeout)
    End Sub

    Overloads Sub PerformOperation(timeout As TimeSpan)
        ' Insert code here.
    End Sub
```

```

    ReadOnly Property OperationTimeout() As TimeSpan
    Get
        Return default tTimeout
    End Get
End Property
End Class
[C#]
class Server
{
    TimeSpan default tTimeout = new TimeSpan(1000);

    void PerformOperation()
    {
        this.PerformOperation(OperationTimeout);
    }

    void PerformOperation(TimeSpan timeout)
    {
        // Insert code here.
    }

    TimeSpan OperationTimeout
    {
        get
        {
            return default tTimeout;
        }
    }
}

```

Types that are not able to resolve time-outs to the resolution of a **TimeSpan** should round the time-out to the nearest interval that can be accommodated. For example, a type that can only wait in one-second increments should round to the nearest second. An exception to this rule is when a value is rounded down to zero. In this case, the time-out should be rounded up to the minimum time-out possible. Rounding away from zero prevents "busy-wait" loops where a zero time-out value causes 100 percent processor utilization.

In addition, it is recommended that you throw an exception when a time-out expires instead of returning an error code. Expiration of a time-out means that the operation could not complete successfully and therefore should be treated and handled as any other run-time error. For more information, see [Error Raising and Handling Guidelines](#).

In the case of an asynchronous operation with a time-out, the callback function should be called and an exception thrown when the results of the operation are first accessed. This is illustrated in the following code example.

VB

```

Sub OnReceiveCompleted(ByVal sender As System.Object, ByVal asyncResult As ReceiveCompletedEventArgs)
    Dim queue As MessageQueue = CType(sender, MessageQueue)
    ' The following code will throw an exception
    ' if BeginReceive has timed out.
    Dim message As Message = queue.EndReceive(asyncResult.AsyncResult)
    Console.WriteLine("Message: " + CStr(message.Body))
    queue.BeginReceive(New TimeSpan(1, 0, 0))
End Sub
[C#]
void OnReceiveCompleted(Object sender, ReceiveCompletedEventArgs asyncResult)
{
    MessageQueue queue = (MessageQueue) sender;
    // The following code will throw an exception
    // if BeginReceive has timed out.
    Message message = queue.EndReceive(asyncResult.AsyncResult);
    Console.WriteLine("Message: " + (string)message.Body);
    queue.BeginReceive(new TimeSpan(1, 0, 0));
}

```

For related information, see [Guidelines for Asynchronous Programming](#).

See Also

[Design Guidelines for Class Library Developers | Error Raising and Handling Guidelines](#)

Security in Class Libraries

.NET Framework 1.1

Class library designers must understand [code access security](#) in order to write secure class libraries. When writing a class library, be aware of two security principles: use permissions to help protect objects, and write fully trusted code. The degree to which these principles apply will depend upon the class you are writing. Some classes, such as the [System.IO.FileStream Class](#), represent objects that need protection with permissions. The implementation of these classes is responsible for checking the permissions of callers and allowing only authorized callers to perform operations for which they have permission. The [System.Security Namespace](#) contains classes that can help you perform these checks in the class libraries that you write. Class library code often is fully trusted or at least highly trusted code. Because class library code often accesses protected resources and unmanaged code, any flaws in the code represent a serious threat to the integrity of the entire security system. To help minimize security threats, follow the guidelines described in this topic when writing class library code. For more information, see [Writing Secure Class Libraries](#).

Protecting Objects with Permissions

[Permissions](#) are defined to help protect specific resources. A class library that performs operations on protected resources must be responsible for enforcing this protection. Before acting on any request on a protected resource, such as deleting a file, class library code first must check that the caller (and usually all callers, by means of a stack walk) has the appropriate delete permission for the resource. If the caller has the permission, the action should be allowed to complete. If the caller does not have the permission, the action should not be allowed to complete and a security exception should be raised. Protection is typically implemented in code with either a [declarative](#) or an [imperative](#) check of the appropriate permissions.

It is important that classes protect resources, not only from direct access, but from all possible kinds of exposure. For example, a cached file object is responsible for checking for file read permissions, even if the actual data is retrieved from a cache in memory and no actual file operation occurs. This is because the effect of handing the data to the caller is the same as if the caller had performed an actual read operation.

Fully Trusted Class Library Code

Many class libraries are implemented as fully trusted code that encapsulates platform-specific functionality as managed objects, such as COM or system APIs. Fully trusted code can expose a weakness to the security of the entire system. However, if class libraries are written correctly with respect to security, placing a heavy security burden on a relatively small set of class libraries and the core runtime security allows the larger body of managed code to acquire the security benefits of these core class libraries.

In a common class library security scenario, a fully trusted class exposes a resource that is protected by a permission; the resource is accessed by a native code API. A typical example of this type of resource is a file. The [File class](#) uses a native API to perform file operations, such as a deletion. The following steps are taken to protect the resource.

1. A caller requests the deletion of file `c:\test.txt` by calling the [File.Delete Method](#).
2. The **Delete** method creates a permission object representing the `delete c:\test.txt` permission.
3. The **File** class's code checks all callers on the stack to see if they have been granted the demanded permission; if not, a security exception is raised.
4. The **File** class asserts FullTrust in order to call native code, because its callers might not have this permission.
5. The **File** class uses a native API to perform the file delete operation.
6. The **File** class returns to its caller, and the file delete request is completed successfully.

Precautions for Highly Trusted Code

Code in a trusted class library is granted permissions that are not available to most application code. In addition, an assembly might contain classes that do not need special permissions but are granted these permissions because the assembly contains other classes that do require them. These situations can expose a security weakness to the system. Therefore, you must be take special care when writing highly or fully trusted code.

Design trusted code so that it can be called by any semi-trusted code on the system without exposing security holes. Resources are normally protected by a stack walk of all callers. If a caller has insufficient permissions, attempted access is blocked. However, any time trusted code asserts a permission, the code takes responsibility for checking for required permissions. Normally, an assert should follow a permission check of the caller as described earlier in this topic. In addition, the number of higher permission asserts should be minimized to reduce the risk of unintended exposure.

Fully trusted code is implicitly granted all other permissions. In addition, it is allowed to violate rules of type safety and object usage. Independent of the protection of resources, any aspect of the programmatic interface that might break type safety or allow access to data not normally available to the caller can lead to a security problem.

Performance

Security checks involve checking the stack for the permissions of all callers. Depending upon the depth of the stack, these operations have the potential to be very expensive. If one operation actually consists of a number of actions at a lower level that require security checks, it might greatly improve performance to check caller permissions once and then assert the necessary permission before performing the actions. The assert will stop the stack walk from propagating further up the stack so that the check will stop there and succeed. This technique typically results in a performance improvement if three or more permission checks can be covered at once.

Summary of Class Library Security Issues

- Any class library that uses protected resources must ensure that it does so only within the permissions of its callers.
- Assertion of permissions should be done only when necessary, and should be preceded by the necessary permission checks.
- To improve performance, aggregate operations that will involve security checks and consider the use of assert to limit stack walks without compromising security.
- Be aware of how a semi-trusted malicious caller might potentially use a class to bypass security.
- Do not assume that code will be called only by callers with certain permissions.
- Do not define non-type-safe interfaces that might be used to bypass security elsewhere.
- Do not expose functionality in a class that allows a semi-trusted caller to take advantage of the higher trust of the class.

See Also

[Design Guidelines for Class Library Developers](#) | [Writing Secure Class Libraries](#) | [Code Access Security](#) | Security and Culture-Aware String Operations

Security and Culture-Aware String Operations

.NET Framework 1.1

The culture-sensitive string operations provided by the .NET Framework can be an advantage to developers creating applications designed to display results to users on a per-culture basis. By default, culture-sensitive methods obtain the culture to use from the current thread's [CultureInfo.CurrentCulture](#) property. For example, the [String.Compare](#) method returns a result that varies by culture due to the differences in sort orders and case mappings used by different cultures. However, culture-sensitive string operations are not always the desired behavior. Using culture-sensitive operations in scenarios where results should be independent of culture can cause code to fail on cultures with [custom case mappings and sorting rules](#), and create security vulnerabilities in your application.

Culture-sensitive string operations can cause security vulnerabilities when the behavior expected by a developer writing a class library differs from the operation's actual behavior on a computer on which the operation runs. These changes in behavior can occur if the culture is changed or if the culture on the computer on which the operation runs differs from the culture the developer used to test the code.

Security Vulnerabilities in Culture-Sensitive String Operations

The unique case-mapping rules for the Turkish alphabet illustrate how a culture-sensitive operation can be used to exploit a security vulnerability in application code. In most Latin alphabets, the character *i* (Unicode 0069) is the lowercase version of the character *I* (Unicode 0049). However, the Turkish alphabet has two versions of the character *I*: one with a dot and one without a dot. In Turkish, the character *I* (Unicode 0049) is considered the uppercase version of a different character *ı* (Unicode 0131). The character *i* (Unicode 0069) is considered the lowercase version of yet another character *İ* (Unicode 0130). As a result, a case-insensitive string comparison of the characters *i* (Unicode 0069) and *I* (Unicode 0049) that succeeds for most cultures fails for the culture "tr-TR" (Turkish in Turkey).

The following code example demonstrates how the result of a case-insensitive **String.Compare** operation performed on the strings "FILE" and "file" differs depending on culture. The comparison returns **true** if the [Thread.CurrentCulture Property](#) is set to "en-US" (English in the United States). The comparison returns **false** if the **CurrentCulture** is set to "tr-TR" (Turkish in Turkey). If an application makes a trust decision based on the results of this **String.Compare** operation, the result of that decision could be subverted by changing the **CurrentCulture**.

VB

```
Imports System
Imports System.Globalization
Imports System.Threading

Public Class TurkishSample
    Public Shared Sub Main()
        ' Set the CurrentCulture property to English in the U.S.
        Thread.CurrentThread.CurrentCulture = New CultureInfo("en-US")
        Console.WriteLine("Culture = {0}", _
            Thread.CurrentThread.CurrentCulture.DisplayName)
        Console.WriteLine("(file == FILE) = {0}", String.Compare("file", _
            "FILE", True) = 0)

        ' Set the CurrentCulture property to Turkish in Turkey.
        Thread.CurrentThread.CurrentCulture = New CultureInfo("tr-TR")
        Console.WriteLine("Culture = {0}", _
            Thread.CurrentThread.CurrentCulture.DisplayName)
        Console.WriteLine("(file == FILE) = {0}", String.Compare("file", _
            "FILE", True) = 0)
    End Sub
End Class

[C#]
using System;
using System.Globalization;
using System.Threading;

public class TurkishSample
{
    public static void Main()
    {
        // Set the CurrentCulture property to English in the U.S.
        Thread.CurrentThread.CurrentCulture = new CultureInfo("en-US");
        Console.WriteLine("Culture = {0}",
            Thread.CurrentThread.CurrentCulture.DisplayName);
        Console.WriteLine("(file == FILE) = {0}", (string.Compare("file",
            "FILE", true) == 0));

        // Set the CurrentCulture property to Turkish in Turkey.
        Thread.CurrentThread.CurrentCulture = new CultureInfo("tr-TR");
        Console.WriteLine("Culture =
            {0}", Thread.CurrentThread.CurrentCulture.DisplayName);
        Console.WriteLine("(file == FILE) = {0}", (string.Compare("file",
            "FILE", true) == 0));
    }
}
```

The following output to the console illustrates how the results vary by culture because the case-insensitive comparison of *i* and *I* evaluates to **true** for the "en-US" culture and **false** for the "tr-TR" culture.

```
Culture = English (United States)
(file == FILE) = True
Culture = Turkish (Turkey)
```

```
(file == FILE) = False
```

For more information about custom sorting rules and case mappings that can cause similar inconsistencies, see [Custom Case Mappings and Sorting Rules](#).

Specifying Culture Explicitly in String Operations

Whether a string operation should be culture-sensitive or culture-insensitive depends on how the application uses the results. String operations that display results to the end user should typically be culture-sensitive. For example, if an application displays a sorted list of localized strings in a list box to the user, you should perform a culture-sensitive sort. Results of string operations that are used internally should typically be culture-insensitive. In general, if you are working with file names, persistence formats, or symbolic information that is not displayed to the end user, results of string operations should not vary by culture. For example, if an application compares a string to determine whether it is a recognized XML tag, the comparison should not be culture-sensitive.

Most .NET Framework methods that perform culture-sensitive string operations by default provide method overloads that allow you to explicitly specify the culture to use by passing a **CultureInfo** parameter. Use these overloads to clearly demonstrate whether a string operation is intended to be culture-sensitive or culture-insensitive. For culture-sensitive operations, specify the **CurrentCulture** property for the **CultureInfo** parameter. To help eliminate security vulnerabilities caused by cultural variations in sorting rules and case mappings, perform culture-insensitive operations by specifying the **CultureInfo.InvariantCulture** property for the **CultureInfo** parameter. This ensures that your code will execute on all computers, regardless of culture, with the same behavior as it does during testing.

Performing Culture-Insensitive String Operations

The following .NET Framework APIs perform culture-sensitive string operations by default. When using these APIs, always use the method overload or class constructor that allows you to explicitly specify the culture to use. Follow the guidelines described in [Specifying Culture Explicitly in String Operations](#) to determine whether you should specify **CurrentCulture** (for culture-sensitive results) or **InvariantCulture** (for culture-insensitive results).

[String.Compare Method](#)

[String.CompareTo Method](#)

[String.ToUpper Method](#)

[String.ToLower Method](#)

[Char.ToUpper Method](#)

[Char.ToLower Method](#)

[CaseInsensitiveComparer Class](#)

[CaseInsensitiveHashCodeProvider Class](#)

[SortedList Class](#)

[ArrayList.Sort Method](#)

[CollectionsUtil.CreateCaseInsensitiveHashtable Method](#)

[Array.Sort Method](#)

[Array.BinarySearch Method](#)

[System.Text.RegularExpressions Namespace](#)

The following topics provide more information about these APIs and examples that demonstrate how to correctly use these them to obtain culture-insensitive results:

- [Performing Culture-Insensitive String Comparisons](#) describes how to use the **String.Compare** and **String.CompareTo** methods to perform culture-insensitive string comparisons.
- [Performing Culture-Insensitive Case Changes](#) describes how to use the **String.ToUpper**, **String.ToLower**, **Char.ToUpper**, and **Char.ToLower** methods to perform culture-insensitive case changes.
- [Performing Culture-Insensitive String Operations in Collections](#) describes how to use the **CaseInsensitiveComparer** class, **CaseInsensitiveHashCodeProvider** class, **SortedList** class, **ArrayList.Sort** method, and **CollectionsUtil.CreateCaseInsensitiveHashtable** method to perform culture-insensitive operations in collections.
- [Performing Culture-Insensitive String Operations in Arrays](#) describes how to use the **Array.Sort** and **Array.BinarySearch** methods to perform culture-insensitive operations in arrays.
- [Performing Culture-Insensitive Operations in the RegularExpressions Namespace](#) describes how to perform culture-insensitive operations using methods in the **RegularExpressions** namespace

Summary of Culture-Aware String Usage Guidelines

Consider the following guidelines when performing culture-aware string operations in your class library code:

- Explicitly pass culture information to all culture-sensitive operations. Specify the **CultureInfo.CurrentCulture** if you want culture-sensitive behavior. Specify the **CultureInfo.InvariantCulture** if you want culture-insensitive behavior.
- Use the **String.Compare** method instead of the **String.CompareTo** method. The **String.Compare** method makes it clear whether you want an operation to be culture-sensitive or culture-insensitive. For code examples that demonstrate how to use the **String.Compare** method, see [Performing Culture-Insensitive String Comparisons](#).
- Provide ways to customize culture for all components that use culture-sensitive operations internally.

See Also

[Security in Class Libraries | Performing Culture-Insensitive String Operations](#)

Threading Design Guidelines

.NET Framework 1.1

The following rules outline the design guidelines for implementing threading:

- Avoid providing static methods that alter static state. In common server scenarios, static state is shared across requests, which means multiple threads can execute that code at the same time. This opens up the possibility for threading bugs. Consider using a design pattern that encapsulates data into instances that are not shared across requests.
- Static state must be thread safe.
- Instance state does not need to be thread safe. By default, class libraries should not be thread safe. Adding locks to create thread-safe code decreases performance, increases lock contention, and creates the possibility for deadlock bugs to occur. In common application models, only one thread at a time executes user code, which minimizes the need for thread safety. For this reason, the .NET Framework class libraries are not thread safe by default. In cases where you want to provide a thread-safe version, provide a static **Synchronized** method that returns a thread-safe instance of a type. For an example, see the [System.Collections.ArrayList.Synchronized Method](#) and the [System.Collections.ArrayList.IsSynchronized Method](#).
- Design your library with consideration for the stress of running in a server scenario. Avoid taking locks whenever possible.
- Be aware of method calls in locked sections. Deadlocks can result when a static method in class A calls static methods in class B and vice versa. If A and B both synchronize their static methods, this will cause a deadlock. You might discover this deadlock only under heavy threading stress.
- Performance issues can result when a static method in class A calls a static method in class A. If these methods are not factored correctly, performance will suffer because there will be a large amount of redundant synchronization. Excessive use of fine-grained synchronization might negatively impact performance. In addition, it might have a significant negative impact on scalability.
- Be aware of issues with the **lock** statement (**SyncLock** in Visual Basic). It is tempting to use the **lock** statement to solve all threading problems. However, the [System.Threading.Interlocked Class](#) is superior for updates that must be atomic. It executes a single **lock** prefix if there is no contention. In a code review, you should watch out for instances like the one shown in the following example.

VB

```
SyncLock Me
    myField += 1
End SyncLock
[C#]
lock(this)
{
    myField++;
}
```

If you replace the previous example with the following one, you will improve performance.

VB

```
System.Threading.Interlocked.Increment(myField)
[C#]
System.Threading.Interlocked.Increment(myField);
```

Another example is to update an object type variable only if it is null (**Nothing** in Visual Basic). You can use the following code to update the variable and make the code thread safe.

VB

```
If x Is Nothing Then
    SyncLock Me
        If x Is Nothing Then
            x = y
        End If
    End SyncLock
End If
[C#]
if (x == null)
{
    lock (this)
    {
        if (x == null)
        {
            x = y;
        }
    }
}
```

You can improve the performance of the previous sample by replacing it with the following code.

VB

```
System.Threading.Interlocked.CompareExchange(x, y, Nothing)
[C#]
System.Threading.Interlocked.CompareExchange(ref x, y, null);
```

- Avoid the need for synchronization if possible. For high traffic pathways, it is best to avoid synchronization. Sometimes the algorithm can be adjusted to tolerate race conditions rather than eliminate them.

See Also

[Design Guidelines for Class Library Developers](#) | [Visual Basic Language Changes](#) | [System.Threading Namespace](#)

Guidelines for Asynchronous Programming

.NET Framework 1.1

Asynchronous programming is a feature supported by many areas of the common language runtime, such as [Remoting](#), [ASP.NET](#), and Windows Forms. Asynchronous programming is a core concept in the .NET Framework. This topic introduces the design pattern for asynchronous programming.

The philosophy behind these guidelines is as follows:

- The client should decide whether a particular call should be asynchronous.
- It is not necessary for a server to do additional programming in order to support its clients' asynchronous behavior. The runtime should be able to manage the difference between the client and server views. As a result, the situation where the server has to implement **IDispatch** and do a large amount of work to support dynamic invocation by clients is avoided.
- The server can choose to explicitly support asynchronous behavior either because it can implement asynchronous behavior more efficiently than a general architecture, or because it wants to support only asynchronous behavior by its clients. It is recommended that such servers follow the design pattern outlined in this document for exposing asynchronous operations.
- Type safety must be enforced.
- The runtime provides the necessary services to support the asynchronous programming model. These services include the following:
 - Synchronization primitives, such as critical sections and [ReaderWriterLock](#) instances.
 - Synchronization constructs such as containers that support the **WaitForMultipleObjects** method.
 - Thread pools.
 - Exposure to the underlying infrastructure, such as [Message](#) and [ThreadPool](#) objects.

See Also

[Design Guidelines for Class Library Developers](#) | [Asynchronous Programming Design Pattern](#)

Asynchronous Programming Design Pattern

.NET Framework 1.1

The following code example demonstrates a server class that factorizes a number.

VB

```
Public Class PrimeFactorizer
    Public Function Factorize(factorizableNum As Long, ByRef primefactor1
        As Long, ByRef primefactor2 As Long) As Boolean
        primefactor1 = 1
        primefactor2 = factorizableNum

        ' Factorize using a low-tech approach.
        Dim i As Integer
        For i = 2 To factorizableNum - 1
            If 0 = factorizableNum Mod i Then
                primefactor1 = i
                primefactor2 = factorizableNum / i
                Exit For
            End If
        Next i
        If 1 = primefactor1 Then
            Return False
        Else
            Return True
        End If
    End Function
End Class
[C#]
public class PrimeFactorizer
{
    public bool Factorize(long factorizableNum,
        ref long primefactor1,
        ref long primefactor2)
    {
        primefactor1 = 1;
        primefactor2 = factorizableNum;

        // Factorize using a low-tech approach.
        for (int i=2; i<factorizableNum; i++)
        {
            if (0 == (factorizableNum % i))
            {
                primefactor1 = i;
                primefactor2 = factorizableNum / i;
                break;
            }
        }
        if (1 == primefactor1 )
            return false;
        else
            return true ;
    }
}
```

The following code example shows a client defining a pattern for asynchronously invoking the Factorize method from the PrimeFactorizer class in the previous example.

VB

```
' Define the delegate.
Delegate Function FactorizingAsyncDelegate(factorizableNum As Long, ByRef
    primefactor1 As Long, ByRef primefactor2 As Long)
End Sub
' Create an instance of the Factorizer.
Dim pf As New PrimeFactorizer()

' Create a delegate on the Factorize method on the Factorizer.
Dim fd As New FactorizingDelegate(pf.Factorize)
[C#]
// Define the delegate.
public delegate bool FactorizingAsyncDelegate(long factorizableNum,
    ref long primefactor1,
    ref long primefactor2);

// Create an instance of the Factorizer.
PrimeFactorizer pf = new PrimeFactorizer();

// Create a delegate on the Factorize method on the Factorizer.
FactorizingDelegate fd = new FactorizingDelegate(pf.Factorize);
```

The compiler will emit the following **FactorizingAsyncDelegate** class after parsing its definition in the first line of the previous example. It will generate the **BeginInvoke** and **EndInvoke** methods.

VB

```
Public Class FactorizingAsyncDelegate
    Inherits Delegate

    Public Function Invoke(factorizableNum As Long, ByRef primefactor1 As
        Long, ByRef primefactor2 As Long) As Boolean
    End Function

    ' Supplied by the compiler.
    Public Function BeginInvoke(factorizableNum As Long, ByRef primefactor1
        As Long, ByRef primefactor2 As Long, cb As AsyncCallback,
        AsyncState As Object) As
        IAsyncResult
    End Function

    ' Supplied by the compiler.
    Public Function EndInvoke(ByRef primefactor1 As Long, ByRef
        primefactor2 As Long, ar As IAsyncResult) As Boolean
    End Function
End Class
```

[C#]

```
public class FactorizingAsyncDelegate : Delegate
{
    public bool Invoke(ulong factorizableNum,
        ref ulong primefactor1, ref ulong primefactor2);

    // Supplied by the compiler.
    public IAsyncResult BeginInvoke(ulong factorizableNum,
        ref unsigned long primefactor1,
        ref unsigned long primefactor2, AsyncCallback cb,
        Object AsyncState);

    // Supplied by the compiler.
    public bool EndInvoke(ref ulong primefactor1,
        ref ulong primefactor2, IAsyncResult ar);
}
```

The interface used as the delegate parameter in the following code example is defined in the .NET Framework Class Library. For more information, see [IAsyncResult Interface](#).

VB

```
Delegate Function AsyncCallback(ar As IAsyncResult)

' Returns true if the asynchronous operation has been completed.
Public Interface IAsyncResult
    ' Handle to block on for the results.
    ReadOnly Property IsCompleted() As Boolean
    ' Get accessor implementation goes here.
    End Property

    ' Caller can use this to wait until operation is complete.
    ReadOnly Property AsyncWaitHandle() As WaitHandle
    ' Get accessor implementation goes here.
    End Property

    ' The delegate object for which the async call was invoked.
    ReadOnly Property AsyncObject() As [Object]
    ' Get accessor implementation goes here.
    End Property

    ' The state object passed in through BeginInvoke.
    ReadOnly Property AsyncState() As [Object]
    ' Get accessor implementation goes here.
    End Property

    ' Returns true if the call completed synchronously.
    ReadOnly Property CompletedSynchronously() As Boolean
    ' Get accessor implementation goes here.
    End Property
End Interface
```

[C#]

```
public delegate AsyncCallback (IAsyncResult ar);

public interface IAsyncResult
{
    // Returns true if the asynchronous operation has completed.
    bool IsCompleted { get; }

    // Caller can use this to wait until operation is complete.
    WaitHandle AsyncWaitHandle { get; }

    // The delegate object for which the async call was invoked.
    Object AsyncObject { get; }
}
```

```

// The state object passed in through BeginInvoke.
Object AsyncState { get; }

// Returns true if the call completed synchronously.
bool CompletedSynchronously { get; }
}

```

Note that the object that implements the [IAsyncResult Interface](#) must be a waitable object and its underlying synchronization primitive should be signaled after the call is canceled or completed. This enables the client to wait for the call to complete instead of polling. The runtime supplies a number of waitable objects that mirror Win32 synchronization primitives, such as [ManualResetEvent](#), [AutoResetEvent](#) and [Mutex](#). It also supplies methods that support waiting for such synchronization objects to become signaled with "any" or "all" semantics. Such methods are context-aware to avoid deadlocks.

The **Cancel** method is a request to cancel processing of the method after the desired time-out period has expired. Note that it is only a request by the client and the server is recommended to honor it. Further, the client should not assume that the server has stopped processing the request completely after receiving notification that the method has been canceled. In other words, the client is recommended to not destroy resources such as file objects, as the server might be actively using them. The **IsCanceled** property will be set to **true** if the call was canceled and the **IsCompleted** property will be set to **true** after the server has completed processing of the call. After the server sets the **IsCompleted** property to **true**, the server cannot use any client-supplied resources outside of the agreed-upon sharing semantics. Thus, it is safe for the client to destroy the resources after the **IsCompleted** property returns **true**.

The **Server** property returns the server object that provided the **IAsyncResult**.

The following code example demonstrates the client-side programming model for invoking the **Factorize** method asynchronously.

C#

```

public class ProcessFactorizeNumber
{
    private long _ulNumber;

    public ProcessFactorizeNumber(long number)
    {
        _ulNumber = number;
    }

    [OneWayAttribute()]
    public void FactorizedResults(IAsyncResult ar)
    {
        long factor1=0, factor2=0;

        // Extract the delegate from the AsyncResult.
        FactorizingAsyncDelegate fd =
            (FactorizingAsyncDelegate)((AsyncResult)ar).AsyncDelegate;
        // Obtain the result.
        fd.EndInvoke(ref factor1, ref factor2, ar);

        // Output the results.
        Console.WriteLine("On CallBack: Factors of {0} : {1} {2}",
            _ulNumber, factor1, factor2);
    }
}

// Async Variation 1.
// The ProcessFactorizeNumber.FactorizedResults callback function
// is called when the call completes.
public void FactorizeNumber1()
{
    // Client code.
    PrimeFactorizer pf = new PrimeFactorizer();
    FactorizingAsyncDelegate fd = new FactorizingAsyncDelegate (pf.Factorize);

    long factorizableNum = 1000589023, temp=0;

    // Create an instance of the class that
    // will be called when the call completes.
    ProcessFactorizeNumber fc =
        new ProcessFactorizeNumber(factorizableNum);
    // Define the AsyncCallback delegate.
    AsyncCallbackDelegate cb = new AsyncCallbackDelegate (fc.FactorizedResults);
    // Any object can be the state object.
    Object state = new Object();

    // Asynchronously invoke the Factorize method on pf.
    // Note: If you have pure out parameters, you do not need the
    // temp variable.
    IAsyncResult ar = fd.BeginInvoke(factorizableNum, ref temp, ref temp,
        cb, state);

    // Proceed to do other useful work.

    // Async Variation 2.
    // Waits for the result.
    // Asynchronously invoke the Factorize method on pf.
    // Note: If you have pure out parameters, you do not need
    // the temp variable.
    public void FactorizeNumber2()
    {

```

```

// Client code.
PrimeFactorizer pf = new PrimeFactorizer();
FactorizingAsyncDelegate fd = new FactorizingAsyncDelegate (pf.Factorize);

long factorizableNum = 1000589023, temp=0;
// Create an instance of the class
// to be called when the call completes.
ProcessFactorizedNumber fc =
    new ProcessFactorizedNumber(factorizableNum);

// Define the AsyncCallback delegate.
AsyncCallback cb = new AsyncCallback(fc.FactorizedResults);

// Any object can be the state object.
Object state = new Object();

// Asynchronously invoke the Factorize method on pf.
IAsyncResult ar = fd.BeginInvoke(factorizableNum, ref temp, ref temp,
    null, null);

ar.AsyncWaitHandle.WaitOne(10000, false);

if(ar.IsCompleted)
{
    int factor1=0, factor2=0;

    // Obtain the result.
    fd.EndInvoke(ref factor1, ref factor2, ar);

    // Output the results.
    Console.WriteLine("Sequential : Factors of {0} : {1} {2}",
        factorizableNum, factor1, factor2);
}
}

```

Note that if `FactorizeCallback` is a context-bound class that requires synchronized or thread-affinity context, the callback function is dispatched through the context dispatcher infrastructure. In other words, the callback function itself might execute asynchronously with respect to its caller for such contexts. These are the semantics of the one-way qualifier on method signatures. Any such method call might execute synchronously or asynchronously with respect to caller, and the caller cannot make any assumptions about completion of such a call when execution control returns to it.

Also, calling **EndInvoke** before the asynchronous operation is complete will block the caller. Calling it a second time with the same `AsyncResult` is undefined.

Summary of Asynchronous Programming Design Pattern

The server splits an asynchronous operation into its two logical parts: the part that takes input from the client and starts the asynchronous operation, and the part that supplies the results of the asynchronous operation to the client. In addition to the input needed for the asynchronous operation, the first part also takes an **AsyncCallbackDelegate** object to be called when the asynchronous operation is completed. The first part returns a waitable object that implements the **IAsyncResult** interface used by the client to determine the status of the asynchronous operation. The server typically also uses the waitable object it returned to the client to maintain any state associated with asynchronous operation. The client uses the second part to obtain the results of the asynchronous operation by supplying the waitable object.

When initiating asynchronous operations, the client can either supply the callback function delegate or not supply it.

The following options are available to the client for completing asynchronous operations:

- Poll the returned **IAsyncResult** object for completion.
- Attempt to complete the operation prematurely, thereby blocking until the operation completes.
- Wait on the **IAsyncResult** object. The difference between this and the previous option is that the client can use time-outs to periodically take back control.
- Complete the operation inside the callback function routine.

One scenario in which both synchronous and asynchronous read and write methods are desirable is the use of file input/output. The following example illustrates the design pattern by showing how the **File** object implements read and write operations.

VB

```

Public Class File
    ' Other methods for this class go here.

    ' Synchronous read method.
    Function Read(buffer() As [Byte], NumToRead As Long) As Long

    ' Asynchronous read method.
    Function BeginRead(buffer() As [Byte], NumToRead As Long, cb As
        AsyncCallbackDelegate) As IAsyncResult
    Function EndRead(ar As IAsyncResult) As Long

    ' Synchronous write method.
    Function Write(buffer() As [Byte], NumToWrite As Long) As Long

    ' Asynchronous write method.
    Function BeginWrite(buffer() As [Byte], NumToWrite As Long, cb As
        AsyncCallbackDelegate) As IAsyncResult
    Function EndWrite(ar As IAsyncResult) As Long
End Class
[C#]
public class File
{
    // Other methods for this class go here.

```

```
// Synchronous read method.
long Read(Byte[] buffer, long NumToRead);

// Asynchronous read method.
IAsyncResult BeginRead(Byte[] buffer, long NumToRead,
    AsyncCallback cb);
long EndRead(IAsyncResult ar);

// Synchronous write method.
long Write(Byte[] buffer, long NumToWrite);

// Asynchronous write method.
IAsyncResult BeginWrite(Byte[] buffer, long NumToWrite,
    AsyncCallback cb);

long EndWrite(IAsyncResult ar);
}
```

The client cannot easily associate state with a given asynchronous operation without defining a new callback function delegate for each operation. This can be fixed by making `Begin` methods, such as `BeginWrite`, take an extra object parameter that represents the state and which is captured in the **`IAsyncResult`**.

See Also

[Design Guidelines for Class Library Developers](#) | [Asynchronous Execution](#) | [IAsyncResult Interface](#)

Naming Guidelines

.NET Framework 1.1

A consistent naming pattern is one of the most important elements of predictability and discoverability in a managed class library. Widespread use and understanding of these naming guidelines should eliminate many of the most common user questions. This topic provides naming guidelines for the .NET Framework types. For each type, you should also take note of some general rules with respect to capitalization styles, case sensitivity and word choice.

In This Section

[Capitalization Styles](#)

Describes the Pascal case, camel case, and uppercase capitalization styles to use to name identifiers in class libraries.

[Case Sensitivity](#)

Describes the case sensitivity guidelines to follow when naming identifiers in class libraries.

[Abbreviations](#)

Describes the guidelines for using abbreviations in type names.

[Word Choice](#)

Lists the keywords to avoid using in type names.

[Avoiding Type Name Confusion](#)

Describes how to avoid using language-specific terminology in order to avoid type name confusion.

[Namespace Naming Guidelines](#)

Describes the guidelines to follow when naming namespaces.

[Class Naming Guidelines](#)

Describes the guidelines to follow when naming classes.

[Interface Naming Guidelines](#)

Describes the guidelines to follow when naming interfaces.

[Attribute Naming Guidelines](#)

Describes the correct way to name an attribute using the Attribute suffix.

[Enumeration Type Naming Guidelines](#)

Describes the guidelines to follow when naming enumerations.

[Static Field Naming Guidelines](#)

Describes the guidelines to follow when naming static fields.

[Parameter Naming Guidelines](#)

Describes the guidelines to follow when naming parameters.

[Method Naming Guidelines](#)

Describes the guidelines to follow when naming methods.

[Property Naming Guidelines](#)

Describes the guidelines to follow when naming properties.

[Event Naming Guidelines](#)

Describes the guidelines to follow when naming events.

Related Sections

[Class Member Usage Guidelines](#)

Describes the guidelines for using properties, events, methods, constructors, fields, and parameters in class libraries.

[Type Usage Guidelines](#)

Describes the guidelines for using classes, value types, delegates, attributes, and nested types in class libraries.

[Design Guidelines for Class Library Developers](#)

Provides naming and usage guidelines for types in the .NET Framework as well as guidelines for implementing common design patterns.

Capitalization Styles

.NET Framework 1.1

Use the following three conventions for capitalizing identifiers.

Pascal case

The first letter in the identifier and the first letter of each subsequent concatenated word are capitalized. You can use Pascal case for identifiers of three or more characters. For example:

```
BackCol or
```

Camel case

The first letter of an identifier is lowercase and the first letter of each subsequent concatenated word is capitalized. For example:

```
backCol or
```

Uppercase

All letters in the identifier are capitalized. Use this convention only for identifiers that consist of two or fewer letters. For example:

```
System. IO
System. Web. UI
```

You might also have to capitalize identifiers to maintain compatibility with existing, unmanaged symbol schemes, where all uppercase characters are often used for enumerations and constant values. In general, these symbols should not be visible outside of the assembly that uses them.

The following table summarizes the capitalization rules and provides examples for the different types of identifiers.

Identifier	Case	Example
Class	Pascal	AppDomain
Enum type	Pascal	ErrorLevel
Enum values	Pascal	FatalError
Event	Pascal	ValueChanged
Exception class	Pascal	WebException Note Always ends with the suffix Exception .
Read-only Static field	Pascal	RedValue
Interface	Pascal	IDisposable Note Always begins with the prefix I .
Method	Pascal	ToString
Namespace	Pascal	System.Drawing
Parameter	Camel	typeName
Property	Pascal	BackColor
Protected instance field	Camel	redValue Note Rarely used. A property is preferable to using a protected instance field.
Public instance field	Pascal	RedValue Note Rarely used. A property is preferable to using a public instance field.

See Also

[Design Guidelines for Class Library Developers](#) | [Naming Guidelines](#)

Case Sensitivity

.NET Framework 1.1

To avoid confusion and guarantee cross-language interoperability, follow these rules regarding the use of case sensitivity:

- Do not use names that require case sensitivity. Components must be fully usable from both case-sensitive and case-insensitive languages. Case-insensitive languages cannot distinguish between two names within the same context that differ only by case. Therefore, you must avoid this situation in the components or classes that you create.
- Do not create two namespaces with names that differ only by case. For example, a case insensitive language cannot distinguish between the following two namespace declarations.

```
namespace ee. cummi ngs;  
namespace Ee. Cummi ngs;
```

- Do not create a function with parameter names that differ only by case. The following example is incorrect.

```
void MyFunction(string a, string A)
```

- Do not create a namespace with type names that differ only by case. In the following example, `Point p` and `POINT p` are inappropriate type names because they differ only by case.

```
System. Wi ndows. Forms. Poi nt p  
System. Wi ndows. Forms. POI NT p
```

- Do not create a type with property names that differ only by case. In the following example, `int Color` and `int COLOR` are inappropriate property names because they differ only by case.

```
int Color {get, set}  
int COLOR {get, set}
```

- Do not create a type with method names that differ only by case. In the following example, `calculate` and `Calculate` are inappropriate method names because they differ only by case.

```
void cal cul ate()  
void Cal cul ate()
```

See Also

[Design Guidelines for Class Library Developers](#) | [Naming Guidelines](#)

Abbreviations

.NET Framework 1.1

To avoid confusion and guarantee cross-language interoperability, follow these rules regarding the use of abbreviations:

- Do not use abbreviations or contractions as parts of identifier names. For example, use `GetWindow` instead of `GetWin`.
- Do not use acronyms that are not generally accepted in the computing field.
- Where appropriate, use well-known acronyms to replace lengthy phrase names. For example, use `UI` for User Interface and `OLAP` for On-line Analytical Processing.
- When using acronyms, use Pascal case or camel case for acronyms more than two characters long. For example, use `HtmlButton` or `htmlButton`. However, you should capitalize acronyms that consist of only two characters, such as `System.IO` instead of `System.Io`.
- Do not use abbreviations in identifiers or parameter names. If you must use abbreviations, use **camel case** for abbreviations that consist of more than two characters, even if this contradicts the standard abbreviation of the word.

See Also

[Design Guidelines for Class Library Developers](#) | [Naming Guidelines](#)

Word Choice

.NET Framework 1.1

Avoid using class names that duplicate commonly used .NET Framework namespaces. For example, do not use any of the following names as a class name: **System**, **Collections**, **Forms**, or **UI**. See the [Class Library](#) for a list of .NET Framework namespaces.

In addition, avoid using identifiers that conflict with the following keywords.

AddHandler	AddressOf	Alias	And	Ansi
As	Assembly	Auto	Base	Boolean
ByRef	Byte	ByVal	Call	Case
Catch	CBool	CByte	CChar	CDate
CDec	CDbI	Char	CInt	Class
CLng	CObj	Const	CShort	CSng
CStr	CType	Date	Decimal	Declare
Default	Delegate	Dim	Do	Double
Each	Else	Elseif	End	Enum
Erase	Error	Event	Exit	ExternalSource
False	Finalize	Finally	Float	For
Friend	Function	Get	GetType	Goto
Handles	If	Implements	Imports	In
Inherits	Integer	Interface	Is	Let
Lib	Like	Long	Loop	Me
Mod	Module	MustInherit	MustOverride	MyBase
MyClass	Namespace	New	Next	Not
Nothing	NotInheritable	NotOverridable	Object	On
Option	Optional	Or	Overloads	Overridable
Overrides	ParamArray	Preserve	Private	Property
Protected	Public	RaiseEvent	ReadOnly	ReDim
Region	REM	RemoveHandler	Resume	Return
Select	Set	Shadows	Shared	Short
Single	Static	Step	Stop	String
Structure	Sub	SyncLock	Then	Throw
To	True	Try	TypeOf	Unicode
Until	volatile	When	While	With
WithEvents	WriteOnly	Xor	eval	extends
instanceof	package	var		

See Also

[Design Guidelines for Class Library Developers](#) | [Naming Guidelines](#)

Avoiding Type Name Confusion

.NET Framework 1.1

Different programming languages use different terms to identify the fundamental managed types. Class library designers must avoid using language-specific terminology. Follow the rules described in this section to avoid type name confusion.

Use names that describe a type's meaning rather than names that describe the type. In the rare case that a parameter has no semantic meaning beyond its type, use a generic name. For example, a class that supports writing a variety of data types into a stream might have the following methods.

VB

```
Sub Write(value As Double);
Sub Write(value As Single);
Sub Write(value As Long);
Sub Write(value As Integer);
Sub Write(value As Short);
[C#]
void Write(double value);
void Write(float value);
void Write(long value);
void Write(int value);
void Write(short value);
```

Do not create language-specific method names, as in the following example.

VB

```
Sub Write(doubleValue As Double);
Sub Write(singleValue As Single);
Sub Write(longValue As Long);
Sub Write(integerValue As Integer);
Sub Write(shortValue As Short);
[C#]
void Write(double doubleValue);
void Write(float floatValue);
void Write(long longValue);
void Write(int intValue);
void Write(short shortValue);
```

In the extremely rare case that it is necessary to create a uniquely named method for each fundamental data type, use a universal type name. The following table lists fundamental data type names and their universal substitutions.

C# type name	Visual Basic type name	JScript type name	Visual C++ type name	llasm.exe representation	Universal type name
sbyte	SByte	sByte	char	int8	SByte
byte	Byte	byte	unsigned char	unsigned int8	Byte
short	Short	short	short	int16	Int16
ushort	UInt16	ushort	unsigned short	unsigned int16	UInt16
int	Integer	int	int	int32	Int32
uint	UInt32	uint	unsigned int	unsigned int32	UInt32
long	Long	long	__int64	int64	Int64
ulong	UInt64	ulong	unsigned __int64	unsigned int64	UInt64
float	Single	float	float	float32	Single
double	Double	double	double	float64	Double
bool	Boolean	boolean	bool	bool	Boolean
char	Char	char	wchar_t	char	Char
string	String	string	String	string	String
object	Object	object	Object	object	Object

For example, a class that supports reading a variety of data types from a stream might have the following methods.

VB

```
ReadDouble()As Double
ReadSingle()As Single
ReadInt64()As Long
```

```
ReadInt32() As Integer
ReadInt16() As Short
[C#]
double ReadDouble();
float ReadSingle();
long ReadInt64();
int ReadInt32();
short ReadInt16();
```

The preceding example is preferable to the following language-specific alternative.

VB

```
ReadDouble() As Double
ReadSingle() As Single
ReadLong() As Long
ReadInteger() As Integer
ReadShort() As Short
[C#]
double ReadDouble();
float ReadFloat();
long ReadLong();
int ReadInt();
short ReadShort();
```

See Also

[Design Guidelines for Class Library Developers](#) | [Common Type System](#)

Namespace Naming Guidelines

.NET Framework 1.1

The general rule for naming namespaces is to use the company name followed by the technology name and optionally the feature and design as follows.

```
CompanyName. TechnologyName[. Feature][. Design]
```

For example:

```
Microsoft. Media  
Microsoft. Media. Design
```

Prefixing namespace names with a company name or other well-established brand avoids the possibility of two published namespaces having the same name. For example, `Microsoft.Office` is an appropriate prefix for the Office Automation Classes provided by Microsoft.

Use a stable, recognized technology name at the second level of a hierarchical name. Use organizational hierarchies as the basis for namespace hierarchies. Name a namespace that contains types that provide design-time functionality for a base namespace with the `.Design` suffix. For example, the [System.Windows.Forms.Design Namespace](#) contains designers and related classes used to design [System.Windows.Forms](#) based applications.

A nested namespace should have a dependency on types in the containing namespace. For example, the classes in the [System.Web.UI.Design](#) depend on the classes in [System.Web.UI](#). However, the classes in **System.Web.UI** do not depend on the classes in **System.Web.UI.Design**.

You should use [Pascal case](#) for namespaces, and separate logical components with periods, as in `Microsoft.Office.PowerPoint`. If your brand employs nontraditional casing, follow the casing defined by your brand, even if it deviates from the prescribed Pascal case. For example, the namespaces `Next.WebObjects` and `ee.cummings` illustrate appropriate deviations from the Pascal case rule.

Use plural namespace names if it is semantically appropriate. For example, use `System.Collections` rather than `System.Collection`. Exceptions to this rule are brand names and abbreviations. For example, use `System.IO` rather than `System.IOs`.

Do not use the same name for a namespace and a class. For example, do not provide both a `Debug` namespace and a `Debug` class.

Finally, note that a namespace name does not have to parallel an assembly name. For example, if you name an assembly `MyCompany.MyTechnology.dll`, it does not have to contain a `MyCompany.MyTechnology` namespace.

See Also

[Design Guidelines for Class Library Developers](#) | [Introduction to the .NET Framework Class Library](#)

Class Naming Guidelines

.NET Framework 1.1

The following rules outline the guidelines for naming classes:

- Use a noun or noun phrase to name a class.
- Use [Pascal case](#).
- Use abbreviations sparingly.
- Do not use a type prefix, such as C_ for class, on a class name. For example, use the class name `FileStream` rather than `CFileStream`.
- Do not use the underscore character (_).
- Occasionally, it is necessary to provide a class name that begins with the letter I, even though the class is not an interface. This is appropriate as long as I is the first letter of an entire word that is a part of the class name. For example, the class name `IdentityStore` is appropriate.
- Where appropriate, use a compound word to name a derived class. The second part of the derived class's name should be the name of the base class. For example, `ApplicationException` is an appropriate name for a class derived from a class named `Exception`, because `ApplicationException` is a kind of `Exception`. Use reasonable judgment in applying this rule. For example, `Button` is an appropriate name for a class derived from `Control`. Although a button is a kind of control, making `Control` a part of the class name would lengthen the name unnecessarily.

The following are examples of correctly named classes.

VB

```
Public Class FileStream
Public Class Button
Public Class String
[C#]
public class FileStream
public class Button
public class String
```

See Also

[Design Guidelines for Class Library Developers](#) | [Base Class Usage Guidelines](#)

Interface Naming Guidelines

.NET Framework 1.1

The following rules outline the naming guidelines for interfaces:

- Name interfaces with nouns or noun phrases, or adjectives that describe behavior. For example, the interface name **IComponent** uses a descriptive noun. The interface name **ICustomAttributeProvider** uses a noun phrase. The name **IPersistable** uses an adjective.
- Use [Pascal case](#).
- Use abbreviations sparingly.
- Prefix interface names with the letter **I**, to indicate that the type is an interface.
- Use similar names when you define a class/interface pair where the class is a standard implementation of the interface. The names should differ only by the letter **I** prefix on the interface name.
- Do not use the underscore character (**_**).

The following are examples of correctly named interfaces.

VB

```
Public Interface IServiceProvider
Public Interface IFormatable
[C#]
public interface IServiceProvider
public interface IFormatable
```

The following code example illustrates how to define the interface **IComponent** and its standard implementation, the class **Component**.

VB

```
Public Interface IComponent
' Implementation code goes here.
End Interface

Public Class Component
Implements IComponent
' Implementation code goes here.
End Class

[C#]
public interface IComponent
{
// Implementation code goes here.
}
public class Component: IComponent
{
// Implementation code goes here.
}
```

See Also

[Design Guidelines for Class Library Developers](#) | [Base Class Usage Guidelines](#)

Attribute Naming Guidelines

.NET Framework 1.1

You should always add the suffix `Attribute` to custom attribute classes. The following is an example of a correctly named attribute class.

VB

```
Public Class ObsoleteAttribute  
[C#]  
public class ObsoleteAttribute{}
```

See Also

[Design Guidelines for Class Library Developers](#)

Enumeration Type Naming Guidelines

.NET Framework 1.1

The enumeration (**Enum**) value type inherits from the [Enum Class](#). The following rules outline the naming guidelines for enumerations:

- Use [Pascal case](#) for **Enum** types and value names.
- Use abbreviations sparingly.
- Do not use an `Enum` suffix on **Enum** type names.
- Use a singular name for most **Enum** types, but use a plural name for **Enum** types that are bit fields.
- Always add the **FlagsAttribute** to a bit field **Enum** type.

See Also

[Design Guidelines for Class Library Developers](#) | [Value Type Usage Guidelines](#) | [Enum Class](#)

Static Field Naming Guidelines

.NET Framework 1.1

The following rules outline the naming guidelines for static fields:

- Use nouns, noun phrases, or abbreviations of nouns to name static fields.
- Use [Pascal case](#).
- Do not use a Hungarian notation prefix on static field names.
- It is recommended that you use static properties instead of public static fields whenever possible.

See Also

[Design Guidelines for Class Library Developers](#) | [Field Usage Guidelines](#)

Parameter Naming Guidelines

.NET Framework 1.1

It is important to carefully follow these parameter naming guidelines because visual design tools that provide context sensitive help and class browsing functionality display method parameter names to users in the designer. The following rules outline the naming guidelines for parameters:

- Use [camel case](#) for parameter names.
- Use descriptive parameter names. Parameter names should be descriptive enough that the name of the parameter and its type can be used to determine its meaning in most scenarios. For example, visual design tools that provide context sensitive help display method parameters to the developer as they type. The parameter names should be descriptive enough in this scenario to allow the developer to supply the correct parameters.
- Use names that describe a parameter's meaning rather than names that describe a parameter's type. Development tools should provide meaningful information about a parameter's type. Therefore, a parameter's name can be put to better use by describing meaning. Use type-based parameter names sparingly and only where it is appropriate.
- Do not use reserved parameters. Reserved parameters are private parameters that might be exposed in a future version if they are needed. Instead, if more data is needed in a future version of your class library, add a new overload for a method.
- Do not prefix parameter names with Hungarian type notation.

The following are examples of correctly named parameters.

VB

```
GetType(typeName As String)As Type
Format(format As String, args() As object)As String
[C#]
Type GetType(string typeName)
string Format(string format, object[] args)
```

See Also

[Design Guidelines for Class Library Developers | Parameter Usage Guidelines](#)

Method Naming Guidelines

.NET Framework 1.1

The following rules outline the naming guidelines for methods:

- Use verbs or verb phrases to name methods.
- Use [Pascal case](#).

The following are examples of correctly named methods.

```
RemoveAll ()  
GetCharArray()  
Invoke()
```

See Also

[Design Guidelines for Class Library Developers](#) | [Method Usage Guidelines](#)

Property Naming Guidelines

.NET Framework 1.1

The following rules outline the naming guidelines for properties:

- Use a noun or noun phrase to name properties.
- Use [Pascal case](#).
- Do not use Hungarian notation.
- Consider creating a property with the same name as its underlying type. For example, if you declare a property named Color, the type of the property should likewise be [Color](#). See the example later in this topic.

The following code example illustrates correct property naming.

VB

```
Public Class SampleClass
    Public Property BackColor As Color
        ' Code for Get and Set accessors goes here.
    End Property
End Class
[C#]
public class SampleClass
{
    public Color BackColor
    {
        // Code for Get and Set accessors goes here.
    }
}
```

The following code example illustrates providing a property with the same name as a type.

VB

```
Public Enum Color
    ' Insert code for Enum here.
End Enum
Public Class Control
    Public Property Color As Color
        Get
            ' Insert code here.
        End Get
        Set
            ' Insert code here.
        End Set
    End Property
End Class
[C#]
public enum Color
{
    // Insert code for Enum here.
}
public class Control
{
    public Color Color
    {
        get { // Insert code here. }
        set { // Insert code here. }
    }
}
```

The following code example is incorrect because the property Color is of type Integer.

VB

```
Public Enum Color
    ' Insert code for Enum here.
End Enum
Public Class Control
    Public Property Color As Integer
        Get
            ' Insert code here.
        End Get
        Set
            ' Insert code here.
        End Set
    End Property
End Class
[C#]
public enum Color { // Insert code for Enum here. }
public class Control
{
    public int Color
```

```
{  
    get { // Insert code here. }  
    set { // Insert code here. }  
}
```

In the incorrect example, it is not possible to refer to the members of the `Color` enumeration. `Color.xxx` will be interpreted as accessing a member that first gets the value of the **Color** property (type **Integer** in Visual Basic or type **int** in C#) and then accesses a member of that value (which would have to be an instance member of **System.Int32**).

See Also

[Design Guidelines for Class Library Developers](#) | [Property Usage Guidelines](#)

Event Naming Guidelines

.NET Framework 1.1

The following rules outline the naming guidelines for events:

- Use [Pascal case](#).
- Do not use Hungarian notation.
- Use an `EventHandler` suffix on event handler names.
- Specify two parameters named *sender* and *e*. The *sender* parameter represents the object that raised the event. The *sender* parameter is always of type **object**, even if it is possible to use a more specific type. The state associated with the event is encapsulated in an instance of an event class named *e*. Use an appropriate and specific event class for the *e* parameter type.
- Name an event argument class with the `EventArgs` suffix.
- Consider naming events with a verb. For example, correctly named event names include **Clicked**, **Painting**, and **DroppedDown**.
- Use a gerund (the "ing" form of a verb) to create an event name that expresses the concept of pre-event, and a past-tense verb to represent post-event. For example, a **Close** event that can be canceled should have a `Closing` event and a `Closed` event. Do not use the `BeforeXxx/AfterXxx` naming pattern.
- Do not use a prefix or suffix on the event declaration on the type. For example, use `Close` instead of `OnClose`.
- In general, you should provide a protected method called `OnXxx` on types with events that can be overridden in a derived class. This method should only have the event parameter *e*, because the sender is always the instance of the type.

The following example illustrates an event handler with an appropriate name and parameters.

VB

```
Public Delegate Sub MouseEventHandler(sender As Object, e As MouseEventArgs)
[C#]
public delegate void MouseEventHandler(object sender, MouseEventArgs e);
```

The following example illustrates a correctly named event argument class.

VB

```
Public Class MouseEventArgs
    Inherits EventArgs
    Dim x As Integer
    Dim y As Integer

    Public Sub New MouseEventArgs(x As Integer, y As Integer)
        me.x = x
        me.y = y
    End Sub

    Public Property X As Integer
        Get
            Return x
        End Get
    End Property

    Public Property Y As Integer
        Get
            Return y
        End Get
    End Property
End Class
[C#]
public class MouseEventArgs : EventArgs
{
    int x;
    int y;
    public MouseEventArgs(int x, int y)
    { this.x = x; this.y = y; }
    public int X { get { return x; } }
    public int Y { get { return y; } }
}
```

See Also

[Design Guidelines for Class Library Developers | Event Usage Guidelines](#)

Class Member Usage Guidelines

.NET Framework 1.1

This topic provides guidelines for using class members in class libraries.

In This Section

[Property Usage Guidelines](#)

Describes the guidelines to follow when using properties in class libraries.

[Event Usage Guidelines](#)

Describes the guidelines to follow when using events in class libraries.

[Method Usage Guidelines](#)

Describes the guidelines to follow when using and overloading methods in class libraries.

[Constructor Usage Guidelines](#)

Describes the guidelines to follow when using constructors in class libraries.

[Field Usage Guidelines](#)

Describes the guidelines to follow when using fields in class libraries.

[Parameter Usage Guidelines](#)

Describes the guidelines to follow when using parameters in class libraries.

Related Sections

[Property Naming Guidelines](#)

Describes the guidelines to follow when naming properties.

[Event Naming Guidelines](#)

Describes the guidelines to follow when naming events.

[Method Naming Guidelines](#)

Describes the guidelines to follow when naming methods.

[Static Field Naming Guidelines](#)

Describes the guidelines to follow when naming static fields.

[Parameter Naming Guidelines](#)

Describes the guidelines to follow when naming parameters.

[Design Guidelines for Class Library Developers](#)

Provides naming and usage guidelines for types in the .NET Framework as well as guidelines for implementing common design patterns.

Property Usage Guidelines

.NET Framework 1.1

Determine whether a property or a method is more appropriate for your needs. For details on choosing between properties and methods, see [Properties vs. Methods](#).

Choose a name for your property based on the recommended [Property Naming Guidelines](#).

When accessing a property using the **set** accessor, preserve the value of the property before you change it. This will ensure that data is not lost if the **set** accessor throws an exception.

Property State Issues

Allow properties to be set in any order. Properties should be stateless with respect to other properties. It is often the case that a particular feature of an object will not take effect until the developer specifies a particular set of properties, or until an object has a particular state. Until the object is in the correct state, the feature is not active. When the object is in the correct state, the feature automatically activates itself without requiring an explicit call. The semantics are the same regardless of the order in which the developer sets the property values or how the developer gets the object into the active state.

For example, a **TextBox** control might have two related properties: **DataSource** and **DataField**. **DataSource** specifies the table name, and **DataField** specifies the column name. Once both properties are specified, the control can automatically bind data from the table into the **Text** property of the control. The following code example illustrates properties that can be set in any order.

VB

```
Dim t As New TextBox()  
t.DataSource = "Publishers"  
t.DataField = "AuthorID"  
' The data-binding feature is now active.  
[C#]  
TextBox t = new TextBox();  
t.DataSource = "Publishers";  
t.DataField = "AuthorID";  
// The data-binding feature is now active.
```

You can set the **DataSource** and **DataField** properties in any order. Therefore, the preceding code is equivalent to the following.

VB

```
Dim t As New TextBox()  
t.DataField = "AuthorID"  
t.DataSource = "Publishers"  
' The data-binding feature is now active.  
[C#]  
TextBox t = new TextBox();  
t.DataField = "AuthorID";  
t.DataSource = "Publishers";  
// The data-binding feature is now active.
```

You can also set a property to null (**Nothing** in Visual Basic) to indicate that the value is not specified.

VB

```
Dim t As New TextBox()  
t.DataField = "AuthorID"  
t.DataSource = "Publishers"  
' The data-binding feature is now active.  
t.DataSource = Nothing  
' The data-binding feature is now inactive.  
[C#]  
TextBox t = new TextBox();  
t.DataField = "AuthorID";  
t.DataSource = "Publishers";  
// The data-binding feature is now active.  
t.DataSource = null;  
// The data-binding feature is now inactive.
```

The following code example illustrates how to track the state of the data binding feature and automatically activate or deactivate it at the appropriate times.

VB

```
Public Class TextBox  
    Private m_dataSource As String  
    Private m_dataField As String  
    Private m_active As Boolean  
  
    Public Property DataSource() As String  
        Get  
            Return m_dataSource  
        End Get  
        Set  
            If value <> m_dataSource Then  
                ' Set the property value first, in case activate fails.
```

```

        m_dataSource = value
        ' Update active state.
        SetActive(( Not (m_dataSource Is Nothing) And Not (m_dataField Is Nothing)))
    End If
End Set
End Property
Public Property DataField() As String
    Get
        Return m_dataField
    End Get
    Set
        If value <> m_dataField Then
            ' Set the property value first, in case activate fails.
            m_dataField = value
            ' Update active state.
            SetActive(( Not (m_dataSource Is Nothing) And Not (m_dataField Is Nothing)))
        End If
    End Set
End Property
Sub SetActive(m_value As Boolean)
    If value <> m_active Then
        If m_value Then
            Activate()
            Text = DataBase.Value(m_dataField)
        Else
            Deactivate()
            Text = ""
        End If
        ' Set active only if successful.
        m_active = value
    End If
End Sub
Sub Activate()
    ' Open database.
End Sub

Sub Deactivate()
    ' Close database.
End Sub
End Class
[C#]
public class TextBox
{
    string dataSource;
    string dataField;
    bool active;

    public string DataSource
    {
        get
        {
            return dataSource;
        }
        set
        {
            if (value != dataSource)
            {
                // Update active state.
                SetActive(value != null && dataField != null);
                dataSource = value;
            }
        }
    }

    public string DataField
    {
        get
        {
            return dataField;
        }
        set
        {
            if (value != dataField)
            {
                // Update active state.
                SetActive(dataSource != null && dataField != null);
                dataField = value;
            }
        }
    }
}
void SetActive(Boolean value)
{
    if (value != active)
    {
        if (value)
        {
            Activate();
            Text = DataBase.Value(dataField);

```

```

    }
    else
    {
        Deactivate();
        Text = "";
    }
    // Set active only if successful.
    active = value;
}
}
void Activate()
{
    // Open database.
}

void Deactivate()
{
    // Close database.
}
}

```

In the preceding example, the following expression determines whether the object is in a state in which the data-binding feature can activate itself.

VB

```

(Not (value Is Nothing) And Not (m_dataField Is Nothing))
[C#]
value != null && dataField != null

```

You make activation automatic by creating a method that determines whether the object can be activated given its current state, and then activates it as necessary.

VB

```

Sub UpdateActive(m_dataSource As String, m_dataField As String)
    SetActive(( Not (m_dataSource Is Nothing) And Not (m_dataField Is Nothing)))
End Sub
[C#]
void UpdateActive(string dataSource, string dataField)
{
    SetActive(dataSource != null && dataField != null);
}

```

If you do have related properties, such as **DataSource** and **DataMember**, you should consider implementing the [ISupportInitialize Interface](#). This will allow the designer (or user) to call the [ISupportInitialize.BeginInit](#) and [ISupportInitialize.EndInit](#) methods when setting multiple properties to allow the component to provide optimizations. In the above example, **ISupportInitialize** could prevent unnecessary attempts to access the database until setup is correctly completed.

The expression that appears in this method indicates the parts of the object model that need to be examined in order to enforce these state transitions. In this case, the **DataSource** and **DataField** properties are affected. For more information on choosing between properties and methods, see [Properties vs. Methods](#).

Raising Property-Changed Events

Components should raise property-changed events if they want to notify consumers when the component's property changes programmatically. The naming convention for a property-changed event is to add the **Changed** suffix to the property name, such as **TextChanged**. For example, a control might raise a **TextChanged** event when its text property changes. You can use a protected helper routine **Raise<Property>Changed**, to raise this event. However, it is probably not worth the overhead to raise a property-changed event for a hash table item addition. The following code example illustrates the implementation of a helper routine on a property-changed event.

VB

```

Class Control
    Inherits Component
    Private m_text As String
    Public Property Text() As String
        Get
            Return m_text
        End Get
        Set
            If Not m_text.Equals(value) Then
                m_text = value
                RaiseTextChanged()
            End If
        End Set
    End Property
End Class
[C#]
class Control: Component
{
    string text;
    public string Text
    {
        get
        {
            return text;
        }
        set
        {
            if (!text.Equals(value))

```

```

        {
            text = value;
            RaiseTextChanged();
        }
    }
}

```

Data binding uses this pattern to allow two-way binding of the property. Without **<Property>Changed** and **Raise<Property>Changed** events, data binding works in one direction; if the database changes, the property is updated. Each property that raises the **<Property>Changed** event should provide metadata to indicate that the property supports data binding.

It is recommended that you raise changing/changed events if the value of a property changes as a result of external forces. These events indicate to the developer that the value of a property is changing or has changed as a result of an operation, rather than by calling methods on the object.

A good example is the **Text** property of an **Edit** control. As a user types information into the control, the property value automatically changes. An event is raised before the value of the property has changed. It does not pass the old or new value, and the developer can cancel the event by throwing an exception. The name of the event is the name of the property followed by the suffix **Changing**. The following code example illustrates a changing event.

VB

```

Class Edit
    Inherits Control

    Public Property Text() As String
        Get
            Return m_text
        End Get
        Set
            If m_text <> value Then
                OnTextChanged(Event.Empty)
                m_text = value
            End If
        End Set
    End Property
End Class
[C#]
class Edit : Control
{
    public string Text
    {
        get
        {
            return text;
        }
        set
        {
            if (text != value)
            {
                OnTextChanged(Event.Empty);
                text = value;
            }
        }
    }
}

```

An event is also raised after the value of the property has changed. This event cannot be canceled. The name of the event is the name of the property followed by the suffix **Changed**. The generic **PropertyChanged** event should also be raised. The pattern for raising both of these events is to raise the specific event from the **OnPropertyChanged** method. The following example illustrates the use of the **OnPropertyChanged** method.

VB

```

Class Edit
    Inherits Control
    Public Property Text() As String
        Get
            Return m_text
        End Get
        Set
            If m_text <> value Then
                OnTextChanged(Event.Empty)
                m_text = value
                RaisePropertyChangedEvent(Edit.ClassInfo, m_text)
            End If
        End Set
    End Property
    Protected Sub OnPropertyChanged(e As PropertyChangedEventArgs)
        If e.PropertyChanged.Equals(Edit.ClassInfo, m_text) Then
            OnTextChanged(Event.Empty)
        End If
        If Not (onPropertyChangedHandler Is Nothing) Then
            onPropertyChangedHandler(Me, e)
        End If
    End Sub
End Class
[C#]

```

```

class Edit : Control
{
    public string Text
    {
        get
        {
            return text;
        }
        set
        {
            if (text != value)
            {
                OnTextChanged(Event.Empty);
                text = value;
                RaisePropertyChangedEvent(Edit.ClassInfo, text);
            }
        }
    }

    protected void OnPropertyChanged(PropertyChangedEventArgs e)
    {
        if (e.PropertyChanged.Equals(Edit.ClassInfo, text))
            OnTextChanged(Event.Empty);
        if (onPropertyChangedHandler != null)
            onPropertyChangedHandler(this, e);
    }
}

```

There are cases when the underlying value of a property is not stored as a field, making it difficult to track changes to the value. When raising the changing event, find all the places that the property value can change and provide the ability to cancel the event. For example, the previous **Edit** control example is not entirely accurate because the **Text** value is actually stored in the window handle (**HWND**). In order to raise the **TextChanged** event, you must examine Windows messages to determine when the text might change, and allow for an exception thrown in **OnTextChanged** to cancel the event. If it is too difficult to provide a changing event, it is reasonable to support only the changed event.

Properties vs. Methods

Class library designers often must decide between implementing a class member as a property or a method. In general, methods represent actions and properties represent data. Use the following guidelines to help you choose between these options.

- Use a property when the member is a logical data member. In the following member declarations, **Name** is a property because it is a logical member of the class.

VB

```

Public Property Name As String
    Get
        Return m_name
    End Get
    Set
        m_name = value
    End Set
End Property
[C#]
public string Name
{
    get
    {
        return name;
    }
    set
    {
        name = value;
    }
}

```

- Use a method when:
 - The operation is a conversion, such as **Object.ToString**.
 - The operation is expensive enough that you want to communicate to the user that they should consider caching the result.
 - Obtaining a property value using the **get** accessor would have an observable side effect.
 - Calling the member twice in succession produces different results.
 - The order of execution is important. Note that a type's properties should be able to be set and retrieved in any order.
 - The member is static but returns a value that can be changed.
 - The member returns an array. Properties that return arrays can be very misleading. Usually it is necessary to return a copy of the internal array so that the user cannot change internal state. This, coupled with the fact that a user can easily assume it is an indexed property, leads to inefficient code. In the following code example, each call to the **Methods** property creates a copy of the array. As a result, $2^n + 1$ copies of the array will be created in the following loop.

VB

```

Dim type As Type = ' Get a type.
Dim i As Integer
For i = 0 To type.Methods.Length - 1
    If type.Methods(i).Name.Equals("text") Then
        ' Perform some operation.
    End If
Next i
[C#]
Type type = // Get a type.
for (int i = 0; i < type.Methods.Length; i++)
{
    if (type.Methods[i].Name.Equals("text"))
    {

```

```

        }
        // Perform some operation.
    }
}

```

The following example illustrates the correct use of properties and methods.

VB

```

Class Connection
    ' The following three members should be properties
    ' because they can be set in any order.
    Property DNSName() As String
        ' Code for get and set accessors goes here.
    End Property
    Property UserName() As String
        ' Code for get and set accessors goes here.
    End Property
    Property Password() As String
        ' Code for get and set accessors goes here.
    End Property
    ' The following member should be a method
    ' because the order of execution is important.
    ' This method cannot be executed until after the
    ' properties have been set.
    Function Execute() As Boolean

```

```

[C#]
class Connection
{
    // The following three members should be properties
    // because they can be set in any order.
    string DNSName {get();set();}
    string UserName {get();set();}
    string Password {get();set();}

    // The following member should be a method
    // because the order of execution is important.
    // This method cannot be executed until after the
    // properties have been set.
    bool Execute ();
}

```

Read-Only and Write-Only Properties

You should use a read-only property when the user cannot change the property's logical data member. Do not use write-only properties.

Indexed Property Usage

Note An indexed property can also be referred to as an indexer.

The following rules outline guidelines for using indexed properties:

- Use an indexed property when the property's logical data member is an array.
- Consider using only integral values or strings for an indexed property. If the design requires other types for the indexed property, reconsider whether it represents a logical data member. If not, use a method.
- Consider using only one index. If the design requires multiple indexes, reconsider whether it represents a logical data member. If not, use a method.
- Use only one indexed property per class, and make it the default indexed property for that class. This rule is enforced by indexer support in the C# programming language.
- Do not use nondefault indexed properties. C# does not allow this.
- Name an indexed property `Item`. For example, see the [DataGrid.Item Property](#). Follow this rule, unless there is a name that is more obvious to users, such as the `chars` property on the **String** class. In C#, indexers are always named `Item`.
- Do not provide an indexed property and a method that are semantically equivalent to two or more overloaded methods. In the following code example, the `Method` property should be changed to `GetMethod(string)` method. Note that this not allowed in C#.

VB

```

' Change the MethodInfo Type.Method property to a method.
Property Type.Method(name As String) As MethodInfo
Function Type.GetMethod(name As String, ignoreCase As Boolean) As MethodInfo
[C#]
// Change the MethodInfo Type.Method property to a method.
MethodInfo Type.Method[string name]
MethodInfo Type.GetMethod (string name, Boolean ignoreCase)
[Visual Basic]
' The MethodInfo Type.Method property is changed to
' the MethodInfo Type.GetMethod method.
Function Type.GetMethod(name As String) As MethodInfo
Function Type.GetMethod(name As String, ignoreCase As Boolean) As MethodInfo
[C#]
// The MethodInfo Type.Method property is changed to
// the MethodInfo Type.GetMethod method.
MethodInfo Type.GetMethod(string name)
MethodInfo Type.GetMethod (string name, Boolean ignoreCase)

```

See Also

[Design Guidelines for Class Library Developers](#) | [Property Naming Guidelines](#) | [Class Member Usage Guidelines](#)

Event Usage Guidelines

.NET Framework 1.1

The following rules outline the usage guidelines for events:

- Choose a name for your event based on the recommended [Event Naming Guidelines](#).
- When you refer to events in documentation, use the phrase, "an event was raised" instead of "an event was fired" or "an event was triggered."
- In languages that support the **void** keyword, use a return type of void for event handlers, as shown in the following C# code example.

```
public delegate void MouseEventHandler(object sender, MouseEventArgs e);
```

- Use strongly typed event data classes when an event conveys meaningful data, such as the coordinates of a mouse click.
- Event classes should extend the [System.EventArgs Class](#), as shown in the following example.

VB

```
Public Class MouseEventArgs
    Inherits EventArgs
    ' Code for the class goes here.
End Class
[C#]
public class MouseEventArgs {}
```

- Use a **protected** (**Protected** in Visual Basic) virtual method to raise each event. This technique is not appropriate for sealed classes, because classes cannot be derived from them. The purpose of the method is to provide a way for a derived class to handle the event using an override. This is more natural than using delegates in situations where the developer is creating a derived class. The name of the method takes the form OnEventName, where EventName is the name of the event being raised. For example:

VB

```
Public Class Button
    Private onClickHandler As ButtonClickHandler
    Protected Overridable Sub OnClick(e As ClickEventArgs)
        ' Call the delegate if non-null.
        If Not (onClickHandler Is Nothing) Then
            onClickHandler(Me, e)
        End If
    End Sub
End Class

[C#]
public class Button
{
    ButtonClickHandler onClickHandler;

    protected virtual void OnClick(ClickEventArgs e)
    {
        // Call the delegate if non-null.
        if (onClickHandler != null)
            onClickHandler(this, e);
    }
}
```

The derived class can choose not to call the base class during the processing of OnEventName. Be prepared for this by not including any processing in the OnEventName method that is required for the base class to work correctly.

- You should assume that an event handler could contain any code. Classes should be ready for the event handler to perform almost any operation, and in all cases the object should be left in an appropriate state after the event has been raised. Consider using a try/finally block at the point in code where the event is raised. Since the developer can perform a callback function on the object to perform other actions, do not assume anything about the object state when control returns to the point at which the event was raised. For example:

VB

```
Public Class Button
    Private onClickHandler As ButtonClickHandler
    Protected Sub DoClick()
        ' Paint button in indented state.
        PaintDown()
        Try
            ' Call event handler.
            OnClick()
        Finally
            ' Window might be deleted in event handler.
            If Not (windowHandle Is Nothing) Then
                ' Paint button in normal state.
                PaintUp()
            End If
        End Try
    End Sub
    Protected Overridable Sub OnClick(e As ClickEvent)
        If Not (onClickHandler Is Nothing) Then
            onClickHandler(Me, e)
        End If
    End Sub
End Class
```

```

    End Sub
End Class
[C#]
public class Button
{
    ButtonClickHandler onClickHandler;

    protected void DoClick()
    {
        // Paint button in indented state.
        PaintDown();
        try
        {
            // Call event handler.
            OnClick();
        }
        finally
        {
            // Window might be deleted in event handler.
            if (windowHandler != null)
                // Paint button in normal state.
                PaintUp();
        }
    }

    protected virtual void OnClick(ClickEventArgs e)
    {
        if (onClickHandler != null)
            onClickHandler(this, e);
    }
}

```

- Use or extend the [System.ComponentModel.CancelEventArgs Class](#) to allow the developer to control the events of an object. For example, the [TreeView](#) control raises a **BeforeLabelEdit** when the user is about to edit a node label. The following code example illustrates how a developer can use this event to prevent a node from being edited.

VB

```

Public Class Form1
    Inherits Form
    Private treeView1 As New TreeView()

    Sub treeView1_BeforeLabelEdit(source As Object, e As NodeLabelEditEventArgs)
        e.CancelEdit = True
    End Sub
End Class
[C#]
public class Form1: Form
{
    TreeView treeView1 = new TreeView();

    void treeView1_BeforeLabelEdit(object source,
        NodeLabelEditEventArgs e)
    {
        e.CancelEdit = true;
    }
}

```

Note that in this case, no error is generated to the user. The label is read-only.

Cancel events are not appropriate in cases where the developer would cancel the operation and return an exception. In these cases, you should raise an exception inside of the event handler in order to cancel. For example, the user might want to write validation logic in an edit control as shown.

VB

```

Public Class Form1
    Inherits Form
    Private edit1 As EditText = New EditText()

    Sub TextChanging(source As Object, e As EventArgs)
        Throw New RuntimeException("Invalid edit")
    End Sub
End Class
[C#]
public class Form1: Form
{
    EditText edit1 = new EditText();

    void TextChanging(object source, EventArgs e)
    {
        throw new RuntimeException("Invalid edit");
    }
}

```

See Also

[Design Guidelines for Class Library Developers](#) | [Event Naming Guidelines](#) | [Class Member Usage Guidelines](#)

Method Usage Guidelines

.NET Framework 1.1

The following rules outline the usage guidelines for methods:

- Choose a name for your event based on the recommended [Method Naming Guidelines](#).
- Do not use Hungarian notation.
- By default, methods are nonvirtual. Maintain this default in situations where it is not necessary to provide virtual methods. For more information about implementing inheritance, see [Base Class Usage Guidelines](#).

Method Overloading Guidelines

Method overloading occurs when a class contains two methods with the same name, but different signatures. This section provides some guidelines for the use of overloaded methods.

- Use method overloading to provide different methods that do semantically the same thing.
- Use method overloading instead of allowing default arguments. Default arguments do not version well and therefore are not allowed in the [Common Language Specification](#) (CLS). The following code example illustrates an overloaded `String.IndexOf` method.

VB

```
Function String.IndexOf(name As String) As Integer
Function String.IndexOf(name As String, startIndex As Integer) As Integer
[C#]
int String.IndexOf (String name);
int String.IndexOf (String name, int startIndex);
```

- Use default values correctly. In a family of overloaded methods, the complex method should use parameter names that indicate a change from the default state assumed in the simple method. For example, in the following code, the first method assumes the search will not be case-sensitive. The second method uses the name `ignoreCase` rather than `caseSensitive` to indicate how the default behavior is being changed.

VB

```
' Method #1: ignoreCase = false.
Function Type.GetMethod(name As String) As MethodInfo
' Method #2: Indicates how the default behavior of method #1
' is being changed.
Function Type.GetMethod(name As String, ignoreCase As Boolean) As MethodInfo
[C#]
// Method #1: ignoreCase = false.
MethodInfo Type.GetMethod(String name);
// Method #2: Indicates how the default behavior of method #1 is being // changed.
MethodInfo Type.GetMethod (String name, Boolean ignoreCase);
```

- Use a consistent ordering and naming pattern for method parameters. It is common to provide a set of overloaded methods with an increasing number of parameters to allow the developer to specify a desired level of information. The more parameters that you specify, the more detail the developer can specify. In the following code example, the overloaded `Execute` method has a consistent parameter order and naming pattern variation. Each of the `Execute` method variations uses the same semantics for the shared set of parameters.

VB

```
Public Class SampleClass
    Private defaultForA As String = "default value for a"
    Private defaultForB As Integer = "42"
    Private defaultForC As Double = "68.90"

    Overloads Public Sub Execute()
        Execute(defaultForA, defaultForB, defaultForC)
    End Sub

    Overloads Public Sub Execute(a As String)
        Execute(a, defaultForB, defaultForC)
    End Sub

    Overloads Public Sub Execute(a As String, b As Integer)
        Execute(a, b, defaultForC)
    End Sub

    Overloads Public Sub Execute(a As String, b As Integer, c As Double)
        Console.WriteLine(a)
        Console.WriteLine(b)
        Console.WriteLine(c)
        Console.WriteLine()
    End Sub
End Class
[C#]
public class SampleClass
{
    readonly string defaultForA = "default value for a";
    readonly int defaultForB = "42";
    readonly double defaultForC = "68.90";

    public void Execute()
    {
        Execute(defaultForA, defaultForB, defaultForC);
    }
}
```

```

    }

    public void Execute (string a)
    {
        Execute(a, default tForB, default tForC);
    }

    public void Execute (string a, int b)
    {
        Execute (a, b, default tForC);
    }

    public void Execute (string a, int b, double c)
    {
        Console.WriteLine(a);
        Console.WriteLine(b);
        Console.WriteLine(c);
        Console.WriteLine();
    }
}

```

Note that the only method in the group that should be virtual is the one that has the most parameters and only when you need extensibility.

- If you must provide the ability to override a method, make only the most complete overload virtual and define the other operations in terms of it. The following example illustrates this pattern.

VB

```

Public Class SampleClass
    Private myString As String

    Public Sub New(str As String)
        Me.myString = str
    End Sub

    Overloads Public Function IndexOf(s As String) As Integer
        Return IndexOf(s, 0)
    End Function

    Overloads Public Function IndexOf(s As String, startIndex As Integer) As Integer
        Return IndexOf(s, startIndex, myString.Length - startIndex)
    End Function

    Overloads Public Overridable Function IndexOf(s As String,
        startIndex As Integer, count As Integer) As Integer
        Return myString.IndexOf(s, startIndex, count)
    End Function
End Class
[C#]
public class SampleClass
{
    private string myString;

    public MyClass(string str)
    {
        this.myString = str;
    }

    public int IndexOf(string s)
    {
        return IndexOf (s, 0);
    }

    public int IndexOf(string s, int startIndex)
    {
        return IndexOf(s, startIndex, myString.Length - startIndex );
    }

    public virtual int IndexOf(string s, int startIndex, int count)
    {
        return myString.IndexOf(s, startIndex, count);
    }
}

```

Methods With Variable Numbers of Arguments

You might want to expose a method that takes a variable number of arguments. A classic example is the **printf** method in the C programming language. For managed class libraries, use the **params** (**ParamArray** in Visual Basic) keyword for this construct. For example, use the following code instead of several overloaded methods.

VB

```

Sub Format(formatString As String, ParamArray args() As Object)
[C#]
void Format(string formatString, params object [] args)

```

You should not use the **VarArgs** or ellipsis (...) calling convention exclusively because the [Common Language Specification](#) does not support it.

For extremely performance-sensitive code, you might want to provide special code paths for a small number of elements. You should only do this if you are going to special case the entire code path (not just create an array and call the more general method). In such cases, the following pattern is recommended as a balance between performance and the cost of specially cased code.

VB

```
Sub Format(formatString As String, arg1 As Object)
Sub Format(formatString As String, arg1 As Object, arg2 As Object)

Sub Format(formatString As String, ParamArray args() As Object)
[C#]
void Format(string formatString, object arg1)
void Format(string formatString, object arg1, object arg2)

void Format(string formatString, params object [] args)
```

See Also

[Design Guidelines for Class Library Developers](#) | [Method Naming Guidelines](#) | [Class Member Usage Guidelines](#) | [Base Class Usage Guidelines](#)

Constructor Usage Guidelines

.NET Framework 1.1

The following rules outline the usage guidelines for constructors:

- Provide a default private constructor if there are only static methods and properties on a class. In the following example, the private constructor prevents the class from being created.

VB

```
NotInheritable Public Class Environment
    ' Private constructor prevents the class from being created.
    Private Sub New()
        ' Code for the constructor goes here.
    End Sub
End Class
[C#]
public sealed class Environment
{
    // Private constructor prevents the class from being created.
    private Environment()
    {
        // Code for the constructor goes here.
    }
}
```

- Minimize the amount of work done in the constructor. Constructors should not do more than capture the constructor parameter or parameters. This delays the cost of performing further operations until the user uses a specific feature of the instance.
- Provide a constructor for every class. If a type is not meant to be created, use a private constructor. If you do not specify a constructor, many programming language (such as C#) implicitly add a default public constructor. If the class is abstract, it adds a protected constructor. Be aware that if you add a nondefault constructor to a class in a later version release, the implicit default constructor will be removed which can break client code. Therefore, the best practice is to always explicitly specify the constructor even if it is a public default constructor.
- Provide a **protected** (**Protected** in Visual Basic) constructor that can be used by types in a derived class.
- You should not provide constructor without parameters for a value type **struct**. Note that many compilers do not allow a **struct** to have a constructor without parameters. If you do not supply a constructor, the runtime initializes all the fields of the **struct** to zero. This makes array and static field creation faster.
- Use parameters in constructors as shortcuts for setting properties. There should be no difference in semantics between using an empty constructor followed by property **set** accessors, and using a constructor with multiple arguments. The following three code examples are equivalent:

VB

```
' Example #1.
Dim SampleClass As New Class()
SampleClass.A = "a"
SampleClass.B = "b"

' Example #2.
Dim SampleClass As New Class("a")
SampleClass.B = "b"

' Example #3.
Dim SampleClass As New Class("a", "b")
[C#]
// Example #1.
Class SampleClass = new Class();
SampleClass.A = "a";
SampleClass.B = "b";

// Example #2.
Class SampleClass = new Class("a");
SampleClass.B = "b";

// Example #3.
Class SampleClass = new Class ("a", "b");
```

- Use a consistent ordering and naming pattern for constructor parameters. A common pattern for constructor parameters is to provide an increasing number of parameters to allow the developer to specify a desired level of information. The more parameters that you specify, the more detail the developer can specify. In the following code example, there is a consistent order and naming of the parameters for all the `SampleClass` constructors.

VB

```
Public Class SampleClass
    Private Const defaultForA As String = "default value for a"
    Private Const defaultForB As String = "default value for b"
    Private Const defaultForC As String = "default value for c"

    Public Sub New()
        MyClass.New(defaultForA, defaultForB, defaultForC)
        Console.WriteLine("New()")
    End Sub

    Public Sub New(a As String)
        MyClass.New(a, defaultForB, defaultForC)
    End Sub
```

```

Public Sub New(a As String, b As String)
    MyClass.New(a, b, default tForC)
End Sub
Public Sub New(a As String, b As String, c As String)
    Me.a = a
    Me.b = b
    Me.c = c
End Sub
End Class
[C#]
public class SampleClass
{
    private const string default tForA = "default t value for a";
    private const string default tForB = "default t value for b";
    private const string default tForC = "default t value for c";

    public MyClass(): this(default tForA, default tForB, default tForC) {}
    public MyClass (string a) : this(a, default tForB, default tForC) {}
    public MyClass (string a, string b) : this(a, b, default tForC) {}
    public MyClass (string a, string b, string c)
}

```

See Also

[Design Guidelines for Class Library Developers](#) | [Class Member Usage Guidelines](#)

Field Usage Guidelines

.NET Framework 1.1

The following rules outline the usage guidelines for fields:

- Do not use instance fields that are **public** or **protected** (**Public** or **Protected** in Visual Basic). If you avoid exposing fields directly to the developer, classes can be versioned more easily because a field cannot be changed to a property while maintaining binary compatibility. Consider providing **get** and **set** property accessors for fields instead of making them public. The presence of executable code in **get** and **set** property accessors allows later improvements, such as creation of an object on demand, upon usage of the property, or upon a property change notification. The following code example illustrates the correct use of private instance fields with **get** and **set** property accessors.

VB

```
Public Structure Point
    Private xValue As Integer
    Private yValue As Integer

    Public Sub New(x As Integer, y As Integer)
        Me.xValue = x
        Me.yValue = y
    End Sub

    Public Property X() As Integer
        Get
            Return xValue
        End Get
        Set
            xValue = value
        End Set
    End Property
    Public Property Y() As Integer
        Get
            Return yValue
        End Get
        Set
            yValue = value
        End Set
    End Property
End Structure
[C#]
public struct Point
{
    private int xValue;
    private int yValue;

    public Point(int x, int y)
    {
        this.xValue = x;
        this.yValue = y;
    }

    public int X
    {
        get
        {
            return xValue;
        }
        set
        {
            xValue = value;
        }
    }

    public int Y
    {
        get
        {
            return yValue;
        }
        set
        {
            yValue = value;
        }
    }
}
```

- Expose a field to a derived class by using a **protected** property that returns the value of the field. This is illustrated in the following code example.

VB

```
Public Class Control
    Inherits Component
    Private handle As Integer

    Protected ReadOnly Property Handle() As Integer
        Get
```

```

        Return handle
    End Get
End Property
End Class
[C#]
public class Control : Component
{
    private int handle;
    protected int Handle
    {
        get
        {
            return handle;
        }
    }
}

```

- Use the **const** (**Const** in Visual Basic) keyword to declare constant fields that will not change. Language compilers save the values of **const** fields directly in calling code.
- Use public static read-only fields for predefined object instances. If there are predefined instances of an object, declare them as public static read-only fields of the object itself. Use **Pascal case** because the fields are public. The following code example illustrates the correct use of public static read-only fields.

VB

```

Public Structure Color
    Public Shared Red As New Color(&HFF)
    Public Shared Green As New Color(&HFF00)
    Public Shared Blue As New Color(&HFF0000)
    Public Shared Black As New Color(&H0)
    Public Shared White As New Color(&FFFFFF)

    Public Sub New(rgb As Integer)
        ' Insert code here.
    End Sub

    Public Sub New(r As Byte, g As Byte, b As Byte)
        ' Insert code here.
    End Sub

    Public ReadOnly Property RedValue() As Byte
        Get
            Return Color.Red
        End Get
    End Property

    Public ReadOnly Property GreenValue() As Byte
        Get
            Return Color.Green
        End Get
    End Property

    Public ReadOnly Property BlueValue() As Byte
        Get
            Return Color.Blue
        End Get
    End Property
End Structure
[C#]
public struct Color
{
    public static readonly Color Red = new Color(0x0000FF);
    public static readonly Color Green = new Color(0x00FF00);
    public static readonly Color Blue = new Color(0xFF0000);
    public static readonly Color Black = new Color(0x000000);
    public static readonly Color White = new Color(0xFFFFFFFF);

    public Color(int rgb)
    { // Insert code here. }
    public Color(byte r, byte g, byte b)
    { // Insert code here. }

    public byte RedValue
    {
        get
        {
            return Color.Red;
        }
    }
    public byte GreenValue
    {
        get
        {
            return Color.Green;
        }
    }
    public byte BlueValue
    {
        get

```

```
    {  
        return Color;  
    }  
}
```

- Spell out all words used in a field name. Use abbreviations only if developers generally understand them. Do not use uppercase letters for field names. The following is an example of correctly named fields.

VB

```
Class SampleClass  
    Private url As String  
    Private destinationUrl As String  
End Class  
[C#]  
class SampleClass  
{  
    string url;  
    string destinationUrl;  
}
```

- Do not use Hungarian notation for field names. Good names describe semantics, not type.
- Do not apply a prefix to field names or static field names. Specifically, do not apply a prefix to a field name to distinguish between static and nonstatic fields. For example, applying a `g_` or `s_` prefix is incorrect.

See Also

[Design Guidelines for Class Library Developers](#) | [Class Member Usage Guidelines](#) | [Static Field Naming Guidelines](#)

Parameter Usage Guidelines

.NET Framework 1.1

The following rules outline the usage guidelines for parameters:

- Check for valid parameter arguments. Perform argument validation for every public or protected method and property **set** accessor. Throw meaningful exceptions to the developer for invalid parameter arguments. Use the [System.ArgumentException Class](#), or a class derived from **System.ArgumentException**. The following example checks for valid parameter arguments and throws meaningful exceptions.

VB

```
Class SampleClass
    Private countValue As Integer
    Private maxValue As Integer = 100

    Public Property Count() As Integer
        Get
            Return countValue
        End Get
        Set
            ' Check for valid parameter.
            If value < 0 Or value >= maxValue Then
                Throw New ArgumentException("value", value,
                    "Value is invalid.")
            End If
            countValue = value
        End Set
    End Property
    Public Sub SelectItem(start As Integer, [end] As Integer)
        ' Check for valid parameter.
        If start < 0 Then
            Throw New ArgumentOutOfRangeException("start", start, "Start
                is invalid.")
        End If
        ' Check for valid parameter.
        If [end] < start Then
            Throw New ArgumentOutOfRangeException("end", [end], "End is
                invalid.")
        End If
        ' Insert code to do other work here.
        Console.WriteLine("Starting at {0}", start)
        Console.WriteLine("Ending at {0}", [end])
    End Sub
End Class
```

```
[C#]
class SampleClass
{
    public int Count
    {
        get
        {
            return count;
        }
        set
        {
            // Check for valid parameter.
            if (count < 0 || count >= MaxValue)
                throw new ArgumentOutOfRangeException(
                    Sys.GetStrin
                        "InvalidArgument", "value", count.ToString());
        }
    }

    public void Select(int start, int end)
    {
        // Check for valid parameter.
        if (start < 0)
            throw new ArgumentException(
                Sys.GetStrin("InvalidArgument", "start", start.ToString());
        // Check for valid parameter.
        if (end < start)
            throw new ArgumentException(
                Sys.GetStrin("InvalidArgument", "end", end.ToString());
    }
}
```

Note that the actual checking does not necessarily have to happen in the **public** or **protected** method itself. It could happen at a lower level in private routines. The main point is that the entire surface area that is exposed to the developer checks for valid arguments.

- Make sure you fully understand the implications of passing parameters by value or by reference. Passing a parameter by value copies the value being passed and has no effect on the original value. The following method example passes parameters by value.

```
public void Add(object value){}
```

Passing a parameter by reference passes the storage location for the value. As a result, changes can be made to the value of the parameter. The following method example passes a parameter by value.

```
public static int Exchange(ref int location, int value){}
```

An output parameter represents the same storage location as the variable specified as the argument in the method invocation. As a result, changes can be made only to the output parameter. The following method example passes an out parameter.

```
[DllImport("Kernel 32.dll")]  
public static extern bool QueryPerformanceCounter(out long value)
```

See Also

[Design Guidelines for Class Library Developers](#) | [Parameter Naming Guidelines](#) | [Class Member Usage Guidelines](#)

Type Usage Guidelines

.NET Framework 1.1

Types are the units of encapsulation in the common language runtime. For a detailed description of the complete list of data types supported by the runtime, see the [Common Type System](#). This section provides usage guidelines for the basic kinds of types.

In This Section

[Base Class Usage Guidelines](#)

Describes how to implement base classes and when to use interfaces instead of classes.

[Value Type Usage Guidelines](#)

Describes how to use the **Struct** and **Enum** value types.

[Delegate Usage Guidelines](#)

Describes how to use delegates for event notifications and callback functions.

[Attribute Usage Guidelines](#)

Describes how to use attributes as declarative information to describe types.

[Nested Type Usage Guidelines](#)

Describes how to use types defined within the scope of another type.

Related Sections

[Naming Guidelines](#)

Describes the guidelines for naming types in class libraries.

[Class Naming Guidelines](#)

Describes the guidelines to follow when naming classes.

[Attribute Naming Guidelines](#)

Describes the correct way to name an attribute using the `Attribute` suffix.

[Design Guidelines for Class Library Developers](#)

Provides naming and usage guidelines for types in the .NET Framework as well as guidelines for implementing common design patterns.

Base Class Usage Guidelines

.NET Framework 1.1

A class is the most common kind of type. A class can be abstract or sealed. An abstract class requires a derived class to provide an implementation. A sealed class does not allow a derived class. It is recommended that you use classes over other types.

Base classes are a useful way to group objects that share a common set of functionality. Base classes can provide a default set of functionality, while allowing customization through extension.

You should explicitly provide a constructor for a class. Compilers commonly add a public default constructor to classes that do not define a constructor. This can be misleading to a user of the class, if your intention is for the class not to be creatable. Therefore, it is best practice to always define at least one constructor for a class. If you do not want it to be creatable, make the constructor private.

You should add extensibility or polymorphism to your design only if you have a clear customer scenario for it. For example, providing an interface for data adapters is difficult and serves no real benefit. Developers will still have to program against each adapter specifically, so there is only marginal benefit from providing an interface. However, you do need to support consistency between all adapters. Although an interface or abstract class is not appropriate in this situation, providing a consistent pattern is very important. You can provide consistent patterns for developers in base classes. Follow these guidelines for creating base classes.

Base Classes vs. Interfaces

An interface type is a specification of a protocol, potentially supported by many object types. Use base classes instead of interfaces whenever possible. From a versioning perspective, classes are more flexible than interfaces. With a class, you can ship Version 1.0 and then in Version 2.0 add a new method to the class. As long as the method is not abstract, any existing derived classes continue to function unchanged.

Because interfaces do not support implementation inheritance, the pattern that applies to classes does not apply to interfaces. Adding a method to an interface is equivalent to adding an abstract method to a base class; any class that implements the interface will break because the class does not implement the new method.

Interfaces are appropriate in the following situations:

- Several unrelated classes want to support the protocol.
- These classes already have established base classes (for example, some are user interface (UI) controls, and some are XML Web services).
- Aggregation is not appropriate or practical.

In all other situations, class inheritance is a better model.

Protected Methods and Constructors

Provide class customization through protected methods. The public interface of a base class should provide a rich set of functionality for the consumer of the class. However, users of the class often want to implement the fewest number of methods possible to provide that rich set of functionality to the consumer. To meet this goal, provide a set of nonvirtual or final public methods that call through to a single protected method that provides implementations for the methods. This method should be marked with the `Implements` suffix. Using this pattern is also referred to as providing a Template method. The following code example demonstrates this process.

VB

```
Public Class SampleClass

    Private x As Integer
    Private y As Integer
    Private width As Integer
    Private height As Integer
    Private specified As BoundsSpecified

    Overloads Public Sub SetBounds(x As Integer, y As Integer, width As Integer, height As Integer)
        SetBoundsImpl(x, y, width, height, Me.specified)
    End Sub

    Overloads Public Sub SetBounds(x As Integer, y As Integer, width As Integer, height As Integer, specified As BoundsSpecified)
        SetBoundsImpl(x, y, width, height, specified)
    End Sub

    Protected Overridable Sub SetBoundsImpl(x As Integer, y As Integer, width As Integer, height As Integer, specified As BoundsSpecified)
        ' Insert code to perform meaningful operations here.
        Me.x = x
        Me.y = y
        Me.width = width
        Me.height = height
        Me.specified = specified
        Console.WriteLine("x {0}, y {1}, width {2}, height {3}, bounds {4}", Me.x, Me.y, Me.width, Me.height, Me.specified)
    End Sub
End Class

[C#]
public class MyClass
{
    private int x;
    private int y;
    private int width;
    private int height;
    BoundsSpecified specified;
}
```

```
public void SetBounds(int x, int y, int width, int height)
{
    SetBoundsImpl(x, y, width, height, this.specificied);
}

public void SetBounds(int x, int y, int width, int height,
    BoundsSpecified specified)
{
    SetBoundsImpl(x, y, width, height, specified);
}

protected virtual void SetBoundsImpl(int x, int y, int width, int
    height, BoundsSpecified specified)
{
    // Add code to perform meaningful operations here.
    this.x = x;
    this.y = y;
    this.width = width;
    this.height = height;
    this.specificied = specified;
}
}
```

Many compilers, such as the C# compiler, insert a **public** or **protected** constructor if you do not. Therefore, for better documentation and readability of your source code, you should explicitly define a **protected** constructor on all abstract classes.

Sealed Class Usage Guidelines

Use sealed classes if there are only static methods and properties on a class.

See Also

[Design Guidelines for Class Library Developers](#) | [Type Usage Guidelines](#) | [Class Naming Guidelines](#)

Value Type Usage Guidelines

.NET Framework 1.1

A value type describes a value that is represented as a sequence of bits stored on the stack. For a description of all the .NET Framework's built-in data types, see [Value Types](#). This section provides guidelines for using the structure (**struct**) and enumeration (**enum**) value types.

Struct Usage Guidelines

It is recommended that you use a **struct** for types that meet any of the following criteria:

- Act like primitive types.
- Have an instance size under 16 bytes.
- Are immutable.
- Value semantics are desirable.

The following example shows a correctly defined structure.

VB

```
Public Structure Int32
    Implements IFormattable
    Implements IComparable
    Public Const MinValue As Integer = -2147483648
    Public Const MaxValue As Integer = 2147483647

    Private intValue As Integer

    Overloads Public Shared Function ToString(i As Integer) As String
        ' Insert code here.
    End Function

    Overloads Public Function ToString(ByVal format As String, ByVal
        formatProvider As IFormatProvider) As String Implements
        IFormattable.ToString
        ' Insert code here.
    End Function

    Overloads Public Overrides Function ToString() As String
        ' Insert code here.
    End Function
    Public Shared Function Parse(s As String) As Integer
        ' Insert code here.
        Return 0
    End Function

    Public Overrides Function GetHashCode() As Integer
        ' Insert code here.
        Return 0
    End Function

    Public Overrides Overloads Function Equals(obj As Object) As Boolean
        ' Insert code here.
        Return False
    End Function

    Public Function CompareTo(obj As Object) As Integer Implements
        IComparable.CompareTo
        ' Insert code here.
        Return 0
    End Function
End Structure
```

```
[C#]
public struct Int32: IComparable, IFormattable
{
    public const int MinValue = -2147483648;
    public const int MaxValue = 2147483647;

    public static string ToString(int i)
    {
        // Insert code here.
    }

    public string ToString(string format, IFormatProvider formatProvider)
    {
        // Insert code here.
    }

    public override string ToString()
    {
        // Insert code here.
    }

    public static int Parse(string s)
    {
        // Insert code here.
    }
}
```

```

        return 0;
    }

    public override int GetHashCode()
    {
        // Insert code here.
        return 0;
    }

    public override bool Equals(object obj)
    {
        // Insert code here.
        return false;
    }

    public int CompareTo(object obj)
    {
        // Insert code here.
        return 0;
    }
}

```

- Do not provide a default constructor for a **struct**. Note that C# does not allow a **struct** to have a default constructor. The runtime inserts a constructor that initializes all the values to a zero state. This allows arrays of structs to be created without running the constructor on each instance. Do not make a **struct** dependent on a constructor being called for each instance. Instances of structs can be created with a zero value without running a constructor. You should also design a **struct** for a state where all instance data is set to zero, false, or null (as appropriate) to be valid.

Enum Usage Guidelines

The following rules outline the usage guidelines for enumerations:

- Do not use an **Enum** suffix on **enum** types.
- Use an **enum** to strongly type parameters, properties, and return types. Always define enumerated values using an **enum** if they are used in a parameter or property. This allows development tools to know the possible values for a property or parameter. The following example shows how to define an enum type.

```

VB
Public Enum FileMode
    Append
    Create
    CreateNew
    Open
    OpenOrCreate
    Truncate
End Enum
[C#]
public enum FileMode
{
    Append,
    Create,
    CreateNew,
    Open,
    OpenOrCreate,
    Truncate
}

```

The following example shows the constructor for a **FileStream** object that uses the **FileMode** enumeration.

```

VB
Public Sub New(ByVal path As String, ByVal mode As FileMode);
[C#]
public FileStream(string path, FileMode mode);

```

- Use an **enum** instead of static final constants.
- Do not use an **enum** for open sets (such as the operating system version).
- Use the **System.FlagsAttribute** Class to create custom attribute for an **enum** only if a bitwise OR operation is to be performed on the numeric values. Use powers of two for the **enum** values so that they can be easily combined. This attribute is applied in the following code example.

```

VB
<Flags(>
Public Enum WatcherChangeTypes
    Created = 1
    Deleted = 2
    Changed = 4
    Renamed = 8
    All = Created Or Deleted Or Changed Or Renamed
End Enum
[C#]
[Flags()]
public enum WatcherChangeTypes
{
    Created = 1,
    Deleted = 2,
    Changed = 4,
    Renamed = 8,
}

```

```
All = Created | Deleted | Changed | Renamed  
};
```

Note An exception to this rule is when encapsulating a Win32 API. It is common to have internal definitions that come from a Win32 header. You can leave these with the Win32 casing, which is usually all capital letters.

- Consider providing named constants for commonly used combinations of flags. Using the bitwise OR is an advanced concept and should not be required for simple tasks. This is illustrated in the following example of an enumeration.

VB

```
<Flags()> _  
Public Enum FileAccess  
    Read = 1  
    Write = 2  
    ReadWrite = Read Or Write  
End Enum  
[C#]  
[Flags()]  
public enum FileAccess  
{  
    Read = 1,  
    Write = 2,  
    ReadWrite = Read | Write,  
}
```

- Use type **Int32** as the underlying type of an **enum** unless either of the following is true:
 - The **enum** represents flags and there are currently more than 32 flags, or the **enum** might grow to have many flags in the future.
 - The type needs to be different from **int** for backward compatibility.
- Do not assume that **enum** arguments will be in the defined range. It is valid to cast any integer value into an **enum** even if the value is not defined in the **enum**. Perform argument validation as illustrated in the following code example.

VB

```
Public Sub SetColor(newColor As Color)  
    If Not [Enum].IsDefined(GetType(Color), newColor) Then  
        Throw New ArgumentOutOfRangeException()  
    End If  
End Sub  
[C#]  
public void SetColor (Color color)  
{  
    if (!Enum.IsDefined (typeof(Color), color)  
        throw new ArgumentOutOfRangeException();  
}
```

See Also

[Design Guidelines for Class Library Developers](#) | [Enumeration Type Naming Guidelines](#) | [Value Types](#) | [Enumerations](#)

Delegate Usage Guidelines

.NET Framework 1.1

A delegate is a powerful tool that allows the managed code object model designer to encapsulate method calls. Delegates are useful for event notifications and callback functions.

Event notifications

Use the appropriate event design pattern for events even if the event is not user interface-related. For more information on using events, see the [Event Usage Guidelines](#).

Callback functions

Callback functions are passed to a method so that user code can be called multiple times during execution to provide customization. Passing a Compare callback function to a sort routine is a classic example of using a callback function. These methods should use the callback function conventions described in [Callback Function Usage](#).

Name end callback functions with the suffix `Callback`.

See Also

[Design Guidelines for Class Library Developers](#) | [Callback Function Usage](#)

Attribute Usage Guidelines

.NET Framework 1.1

The .NET Framework enables developers to invent new kinds of declarative information, to specify declarative information for various program entities, and to retrieve attribute information in a run-time environment. For example, a framework might define a `HelpAttribute` attribute that can be placed on program elements such as classes and methods to provide a mapping from program elements to their documentation. New kinds of declarative information are defined through the declaration of attribute classes, which might have positional and named parameters. For more information about attributes, see [Writing Custom Attributes](#).

The following rules outline the usage guidelines for attribute classes:

- Add the `Attribute` suffix to custom attribute classes, as shown in the following example.

VB

```
Public Class ObsoleteAttribute()  
[C#]  
public class ObsoleteAttribute()  

```

- Specify `AttributeUsage` on your attributes to define their usage precisely, as shown in the following example.

VB

```
<AttributeUsage(AttributeTargets.All, Inherited := False, AllowMultiple := True)> _  
  
Public Class ObsoleteAttribute  
    Inherits Attribute  
    ' Insert code here.  
End Class  
[C#]  
[AttributeUsage(AttributeTargets.All, Inherited = false, AllowMultiple = true)]  
public class ObsoleteAttribute: Attribute {  

```

- Seal attribute classes whenever possible, so that classes cannot be derived from them.
- Use positional arguments (constructor parameters) for required parameters. Provide a read-only property with the same name as each positional argument, but change the case to differentiate between them. This allows access to the argument at runtime.
- Use named arguments (read/write properties) for optional parameters. Provide a read/write property with the same name as each named argument, but change the case to differentiate between them.
- Do not define a parameter with both named and positional arguments. The following example illustrates this pattern.

VB

```
Public Class NameAttribute  
    Inherits Attribute  
  
    ' This is a positional argument.  
    Public Sub New(username As String)  
        ' Implement code here.  
    End Sub  
  
    Public ReadOnly Property UserName() As String  
        Get  
            Return UserName  
        End Get  
    End Property  
  
    ' This is a named argument.  
    Public Property Age() As Integer  
        Get  
            Return Age  
        End Get  
        Set  
            Age = value  
        End Set  
    End Property  
End Class  
[C#]  
public class NameAttribute: Attribute  
{  
    // This is a positional argument.  
    public NameAttribute (string username)  
    {  
        // Implement code here.  
    }  
    public string UserName  
    {  
        get  
        {  
            return UserName;  
        }  
    }  
    // This is a named argument.  
    public int Age  
    {  
        get  

```

```
    {  
        return Age;  
    }  
    set  
    {  
        Age = value;  
    }  
}
```

See Also

[Design Guidelines for Class Library Developers](#) | [Attribute Naming Guidelines](#)

Nested Type Usage Guidelines

.NET Framework 1.1

A nested type is a type defined within the scope of another type. Nested types are very useful for encapsulating implementation details of a type, such as an enumerator over a collection, because they can have access to private state.

Public nested types should be used rarely. Use them only in situations where both of the following are true:

- The nested type (inner type) logically belongs to the containing type (outer type).

The following examples illustrates how to define types with and without nested types:

VB

```
' With nested types.
ListBox.SelectedObjectCollection
' Without nested types.
ListBoxSelectedObjectCollection

' With nested types.
RichTextBox.ScrollBars
' Without nested types.
RichTextBoxScrollBar

[C#]
// With nested types.
ListBox.SelectedObjectCollection
// Without nested types.
ListBoxSelectedObjectCollection

// With nested types.
RichTextBox.ScrollBars
// Without nested types.
RichTextBoxScrollBar
```

Do not use nested types if the following are true:

- The type must be instantiated by client code. If a type has a public constructor, it probably should not be nested. The rationale behind this guideline is that if a nested type can be instantiated, it indicates that the type has a place in the library on its own. You can create it, use it, and destroy it without using the outer type. Therefore, it should not be nested. An inner type should not be widely reused outside of the outer type without a relationship to the outer type.
- References to the type are commonly declared in client code.

See Also

[Design Guidelines for Class Library Developers](#)

Guidelines for Exposing Functionality to COM

.NET Framework 1.1

The common language runtime provides rich support for interoperating with COM components. A COM component can be used from within a managed type and a managed instance can be used by a COM component. This support is the key to moving unmanaged code to managed code one piece at a time; however, it does present some issues for class library designers. In order to fully expose a managed type to COM clients, the type must expose functionality in a way that is supported by COM and abides by the COM versioning contract.

Mark managed class libraries with the [ComVisibleAttribute](#) attribute to indicate whether COM clients can use the library directly or whether they must use a wrapper that shapes the functionality so that they can use it.

Types and interfaces that must be used directly by COM clients, such as to host in an unmanaged container, should be marked with the **ComVisible(true)** attribute. The transitive closure of all types referenced by exposed types should be explicitly marked as **ComVisible(true)**; if not, they will be exposed as **IUnknown**.

Note Members of a type can also be marked as **ComVisible(false)**; this reduces exposure to COM and therefore reduces the restrictions on what a managed type can use.

Types marked with the **ComVisible(true)** attribute cannot expose functionality exclusively in a way that is not usable from COM. Specifically, COM does not support static methods or parameterized constructors. Test the type's functionality from COM clients to ensure correct behavior. Make sure that you understand the registry impact for making all types cocreateable.

Marshal By Reference

Marshal-by-reference objects are [Remotable Objects](#). Object remoting applies to three kinds of types:

- Types whose instances are copied when they are marshaled across an AppDomain boundary (on the same computer or a different computer). These types must be marked with the **Serializable** attribute.
- Types for which the runtime creates a transparent proxy when they are marshaled across an AppDomain boundary (on the same computer or a different computer). These types must ultimately be derived from [System.MarshalByRefObject Class](#).
- Types that are not marshaled across AppDomains at all. This is the default.

Follow these guidelines when using marshal by reference:

- By default, instances should be marshal-by-value objects. This means that their types should be marked as **Serializable**.
- Component types should be marshal-by-reference objects. This should already be the case for most components, because the common base class, [System.Component Class](#), is a marshal-by-reference class.
- If the type encapsulates an operating system resource, it should be a marshal-by-reference object. If the type implements the [IDisposable Interface](#) it will very likely have to be marshaled by reference. [System.IO.Stream](#) derives from [MarshalByRefObject](#). Most streams, such as FileStreams and NetworkStreams, encapsulate external resources, so they should be marshal-by-reference objects.
- Instances that simply hold state should be marshal-by-value objects (such as a **DataSet**).
- Special types that cannot be called across an AppDomain (such as a holder of static utility methods) should not be marked as **Serializable**.

See Also

[Design Guidelines for Class Library Developers](#) | [System.MarshalByRefObject Class](#) | [Exposing .NET Framework Components to COM](#)

Error Raising and Handling Guidelines

.NET Framework 1.1

The following rules outline the guidelines for raising and handling errors:

- All code paths that result in an exception should provide a method to check for success without throwing an exception. For example, to avoid a **FileNotFoundException** you can call **File.Exists**. This might not always be possible, but the goal is that under normal execution no exceptions should be thrown.
- End **Exception** class names with the **exception** suffix as in the following code example.

VB

```
Public Class FileNotFoundException
    Inherits Exception
    ' Implementation code goes here.
End Class
[C#]
public class FileNotFoundException : Exception
{
    // Implementation code goes here.
}
```

- Use the common constructors shown in the following code example when creating exception classes.

VB

```
Public Class XxxException
    Inherits ApplicationException

    Public Sub New()
        ' Implementation code goes here.
    End Sub

    Public Sub New(message As String)
        ' Implementation code goes here.
    End Sub

    Public Sub New(message As String, inner As Exception)
        ' Implementation code goes here.
    End Sub

    Public Sub New(info As SerializationInfo, context As StreamingContext)
        ' Implementation code goes here.
    End Sub
End Class
[C#]
public class XxxException : ApplicationException
{
    public XxxException() { ... }
    public XxxException(string message) { ... }
    public XxxException(string message, Exception inner) { ... }
    public XxxException(SerializationInfo info, StreamingContext context) { ... }
}
```

- In most cases, use the predefined exception types. Only define new exception types for programmatic scenarios, where you expect users of your class library to catch exceptions of this new type and perform a programmatic action based on the exception type itself. This is in lieu of parsing the exception string, which would negatively impact performance and maintenance.

For example, it makes sense to define a **FileNotFoundException** because the developer might decide to create the missing file. However, a **FileIOException** is not something that would typically be handled specifically in code.

- Do not derive all new exceptions directly from the base class **SystemException**. Inherit from **SystemException** only when creating new exceptions in System namespaces. Inherit from **ApplicationException** when creating new exceptions in other namespaces.
- Group new exceptions derived from **SystemException** or **ApplicationException** by namespace. For example, all **System.IO** exceptions are grouped under **IOException** (derived from **SystemException**) and all Microsoft.Media exceptions could be grouped under **MediaException** (derived from **ApplicationException**).
- Use a localized description string in every exception. When the user sees an error message, it will be derived from the description string of the exception that was thrown, and never from the exception class.
- Create grammatically correct error messages with punctuation. Each sentence in the description string of an exception should end in a period. Code that generically displays an exception message to the user does not have to handle the case where a developer forgot the final period.
- Provide exception properties for programmatic access. Include extra information (other than the description string) in an exception only when there is a programmatic scenario where that additional information is useful. You should rarely need to include additional information in an exception.
- Do not expose privileged information in exception messages. Information such as paths on the local file system is considered privileged information. Malicious code could use this information to gather private user information from the computer.
- Do not use exceptions for normal or expected errors, or for normal flow of control.
- You should return null for extremely common error cases. For example, a **File.Open** command returns a null reference if the file is not found, but throws an exception if the file is locked.
- Design classes so that in the normal course of use an exception will never be thrown. In the following code example, a **FileStream** class exposes another way of determining if the end of the file has been reached to avoid the exception that will be thrown if the developer reads past the end of the file.

VB

```
Class FileRead
    Sub Open()
        Dim stream As FileStream = File.Open("myfile.txt", FileMode.Open)
        Dim b As Byte

        ' ReadByte returns -1 at end of file.
    End Sub
End Class
```

```

        While b = stream.ReadByte() <> true
            ' Do something.
        End While
    End Sub
End Class
[C#]
class FileRead
{
    void Open()
    {
        FileStream stream = File.Open("myfile.txt", FileMode.Open);
        byte b;

        // ReadByte returns -1 at end of file.
        while ((b = stream.ReadByte()) != true)
        {
            // Do something.
        }
    }
}

```

- Throw the [InvalidOperationException](#) exception if a call to a property **set** accessor or method is not appropriate given the object's current state.
- Throw an [ArgumentException](#) or create an exception derived from this class if invalid parameters are passed or detected.
- Be aware that the stack trace starts at the point where an exception is thrown, not where it is created with the **new** operator. Consider this when deciding where to throw an exception.
- Use the exception builder methods. It is common for a class to throw the same exception from different places in its implementation. To avoid repetitive code, use helper methods that create the exception using the **new** operator and return it. The following code example shows how to implement a helper method.

C#

```

class File
{
    string fileName;
    public byte[] Read(int bytes)
    {
        if (!ReadFile(handle, bytes))
            throw new FileNotFoundException();
    }

    FileNotFoundException NewFileNotFoundException()
    {
        string description =
            // Build localized string, include fileName.
        return new FileNotFoundException(description);
    }
}

```

- Throw exceptions instead of returning an error code or HRESULT.
- Throw the most specific exception possible.
- Create meaningful message text for exceptions, targeted at the developer.
- Set all fields on the exception you use.
- Use Inner exceptions (chained exceptions). However, do not catch and re-throw exceptions unless you are adding additional information or changing the type of the exception.
- Do not create methods that throw [NullReferenceException](#) or [IndexOutOfRangeException](#).
- Perform argument checking on protected (Family) and internal (Assembly) members. Clearly state in the documentation if the protected method does not do argument checking. Unless otherwise stated, assume that argument checking is performed. There might, however, be performance gains in not performing argument checking.
- Clean up any side effects when throwing an exception. Callers should be able to assume that there are no side effects when an exception is thrown from a function. For example, if a **Hashtable.Insert** method throws an exception, the caller can assume that the specified item was not added to the **Hashtable**.

Standard Exception Types

The following table lists the standard exceptions provided by the runtime and the conditions for which you should create a derived class.

Exception type	Base type	Description	Example
Exception	Object	Base class for all exceptions.	None (use a derived class of this exception).
SystemException	Exception	Base class for all runtime-generated errors.	None (use a derived class of this exception).
IndexOutOfRangeException	SystemException	Thrown by the runtime only when an array is indexed improperly.	Indexing an array outside of its valid range: arr[arr.Length+1]
NullReferenceException	SystemException	Thrown by the runtime only when a null object is referenced.	object o = null; o.ToString();
InvalidOperationException	SystemException	Thrown by methods when in an invalid state.	Calling Enumerator.GetNext() after removing an Item from the underlying collection.
ArgumentException	SystemException	Base class for all argument	None (use a derived class of

		exceptions.	this exception).
ArgumentNullException	ArgumentException	Thrown by methods that do not allow an argument to be null.	String s = null; "Calculate".IndexOf(s);
ArgumentOutOfRangeException	ArgumentException	Thrown by methods that verify that arguments are in a given range.	String s = "string"; s.Chars[9];
ExternalException	SystemException	Base class for exceptions that occur or are targeted at environments outside of the runtime.	None (use a derived class of this exception).
COMException	ExternalException	Exception encapsulating COM Hresult information.	Used in COM interop.
SEHException	ExternalException	Exception encapsulating Win32 structured Exception Handling information.	Used in unmanaged code Interop.

Wrapping Exceptions

Errors that occur at the same layer as a component should throw an exception that is meaningful to target users. In the following code example, the error message is targeted at users of the **TextReader** class, attempting to read from a stream.

VB

```

Public Class TextReader
    Public Function ReadLine() As String
        Try
            ' Read a line from the stream.
        Catch e As Exception
            Throw New IOException("Could not read from stream", e)
        End Try
    End Function
End Class
[C#]
public class TextReader
{
    public string ReadLine()
    {
        try
        {
            // Read a line from the stream.
        }
        catch (Exception e)
        {
            throw new IOException ("Could not read from stream", e);
        }
    }
}

```

See Also

[Design Guidelines for Class Library Developers | Handling and Throwing Exceptions](#)

Array Usage Guidelines

.NET Framework 1.1

For a general description of arrays and array usage see [Arrays](#), and [System.Array Class](#).

Arrays vs. Collections

Class library designers might need to make difficult decisions about when to use an array and when to return a collection. Although these types have similar usage models, they have different performance characteristics. In general, you should use a collection when **Add**, **Remove**, or other methods for manipulating the collection are supported.

For more information on using collections, see [Grouping Data in Collections](#).

Array Usage

Do not return an internal instance of an array. This allows calling code to change the array. The following example demonstrates how the array `badChars` can be changed by any code that accesses the `Path` property even though the property does not implement the set accessor.

VB

```
Imports System
Imports System.Collections
Imports Microsoft.VisualBasic

Public Class ExampleClass
    NotInherited Public Class Path
        Private Sub New()
            End Sub

        Private Property Path
            Get
                End Get
            Set
                End Set
        End Property

        Private Shared badChars() As Char = {Chr(34), "<", ">"c}

        Public Shared Function GetInvalidPathChars() As Char()
            Return badChars
        End Function

    End Class

    Public Shared Sub Main()
        ' The following code displays the elements of the
        ' array as expected.
        Dim c As Char
        For Each c In Path.GetInvalidPathChars()
            Console.WriteLine(c)
        Next c
        Console.WriteLine()

        ' The following code sets all the values to A.
        Path.GetInvalidPathChars()(0) = "A"c
        Path.GetInvalidPathChars()(1) = "A"c
        Path.GetInvalidPathChars()(2) = "A"c

        ' The following code displays the elements of the array to the
        ' console. Note that the values have changed.
        For Each c In Path.GetInvalidPathChars()
            Console.WriteLine(c)
        Next c
    End Sub
End Class
```

[C#]

```
using System;
using System.Collections;

public class ExampleClass
{
    public sealed class Path
    {
        private Path(){}
        private static char[] badChars = {'\"", '<', '>'};
        public static char[] GetInvalidPathChars()
        {
            return badChars;
        }
    }

    public static void Main()
    {
        // The following code displays the elements of the
        // array as expected.
        foreach(char c in Path.GetInvalidPathChars())
```

```

    {
        Console.WriteLine(c);
    }
    Console.WriteLine();

    // The following code sets all the values to A.
    Path.GetInvalidPathChars()[0] = 'A';
    Path.GetInvalidPathChars()[1] = 'A';
    Path.GetInvalidPathChars()[2] = 'A';

    // The following code displays the elements of the array to the
    // console. Note that the values have changed.
    foreach(char c in Path.GetInvalidPathChars())
    {
        Console.WriteLine(c);
    }
}

```

You can correct the problem in the preceding example by making the `badChars` collection **readonly** (**ReadOnly** in Visual Basic). Alternately, you can clone the `badChars` collection before returning. The following example demonstrates how to modify the `GetInvalidPathChars` method to return a clone of the `badChars` collection.

VB

```

Public Shared Function GetInvalidPathChars() As Char()
    Return CType(badChars.Clone(), Char())
End Function
[C#]
public static char[] GetInvalidPathChars()
{
    return (char[])badChars.Clone();
}

```

Do not use **readonly** (**ReadOnly** in Visual Basic) fields of arrays. If you do, the array is readonly and cannot be changed, but the elements in the array can be changed. The following example demonstrates how the elements of the readonly array `InvalidPathChars` can be changed.

C#

```

public sealed class Path
{
    private Path(){}
    public static readonly char[] InvalidPathChars = {'\\', '<', '>', '|'}
}
//The following code can be used to change the values in the array.
Path.InvalidPathChars[0] = 'A';

```

Using Indexed Properties in Collections

You should use an indexed property only as a default member of a collection class or interface. Do not create families of functions in noncollection types. A pattern of methods, such as **Add**, **Item**, and **Count**, signal that the type should be a collection.

Array Valued Properties

You should use collections to avoid code inefficiencies. In the following code example, each call to the `myObj` property creates a copy of the array. As a result, $2^n + 1$ copies of the array will be created in the following loop.

VB

```

Dim i As Integer
For i = 0 To obj.myObj.Count - 1
    DoSomething(obj.myObj(i))
Next i
[C#]
for (int i = 0; i < obj.myObj.Count; i++)
    DoSomething(obj.myObj[i]);

```

For more information, see the [Properties vs. Methods](#) topic.

Returning Empty Arrays

String and **Array** properties should never return a null reference. Null can be difficult to understand in this context. For example, a user might assume that the following code will work.

VB

```

Public Sub DoSomething()
    Dim s As String = SomeOtherFunc()
    If s.Length > 0 Then
        ' Do something else.
    End If
End Sub
[C#]
public void DoSomething()
{
    string s = SomeOtherFunc();
    if (s.Length > 0)
    {

```

```
} // Do something else.  
}
```

The general rule is that null, empty string (""), and empty (0 item) arrays should be treated the same way. Return an empty array instead of a null reference.

See Also

[Design Guidelines for Class Library Developers](#) | [System.Array Class](#).

Operator Overloading Usage Guidelines

.NET Framework 1.1

The following rules outline the guidelines for operator overloading:

- Define operators on value types that are logical built-in language types, such as the [System.Decimal Structure](#).
- Provide operator-overloading methods only in the class in which the methods are defined. The C# compiler enforces this guideline.
- Use the names and signature conventions described in the [Common Language Specification \(CLS\)](#). The C# compiler does this for you automatically.
- Use operator overloading in cases where it is immediately obvious what the result of the operation will be. For example, it makes sense to be able to subtract one Time value from another Time value and get a TimeSpan. However, it is not appropriate to use the **or** operator to create the union of two database queries, or to use **shift** to write to a stream.
- Overload operators in a symmetric manner. For example, if you overload the equality operator (==), you should also overload the not equal operator(!=).
- Provide alternate signatures. Most languages do not support operator overloading. For this reason, it is a CLS requirement for all types that overload operators to include a secondary method with an appropriate domain-specific name that provides the equivalent functionality. It is a Common Language Specification (CLS) requirement to provide this secondary method. The following example is CLS-compliant.

C#

```
public struct DateTime
{
    public static TimeSpan operator -(DateTime t1, DateTime t2) { }
    public static TimeSpan Subtract(DateTime t1, DateTime t2) { }
}
```

The following table contains a list of operator symbols and the corresponding alternative methods and operator names.

C++ operator symbol	Name of alternative method	Name of operator
Not defined	ToXxx or FromXxx	op_Implicit
Not defined	ToXxx or FromXxx	op_Explicit
+ (binary)	Add	op_Addition
- (binary)	Subtract	op_Subtraction
* (binary)	Multiply	op_Multiply
/	Divide	op_Division
%	Mod	op_Modulus
^	Xor	op_ExclusiveOr
& (binary)	BitwiseAnd	op_BitwiseAnd
	BitwiseOr	op_BitwiseOr
&&	And	op_LogicalAnd
	Or	op_LogicalOr
=	Assign	op_Assign
<<	LeftShift	op_LeftShift
>>	RightShift	op_RightShift
Not defined	LeftShift	op_SignedRightShift
Not defined	RightShift	op_UnsignedRightShift
==	Equals	op_Equality
>	Compare	op_GreaterThan
<	Compare	op_LessThan
!=	Compare	op_Inequality
>=	Compare	op_GreaterThanOrEqual
<=	Compare	op_LessThanOrEqual
*=	Multiply	op_MultiplicationAssignment
-=	Subtract	op_SubtractionAssignment

^ =	Xor	op_ExclusiveOrAssignment
< < =	LeftShift	op_LeftShiftAssignment
% =	Mod	op_ModulusAssignment
+ =	Add	op_AdditionAssignment
& =	BitwiseAnd	op_BitwiseAndAssignment
 =	BitwiseOr	op_BitwiseOrAssignment
,	None assigned	op_Comma
/ =	Divide	op_DivisionAssignment
--	Decrement	op_Decrement
++	Increment	op_Increment
- (unary)	Negate	op_UnaryNegation
+ (unary)	Plus	op_UnaryPlus
~	OnesComplement	op_OnesComplement

See Also

[Design Guidelines for Class Library Developers](#)

Guidelines for Casting Types

.NET Framework 1.1

The following rules outline the usage guidelines for casts:

- Do not allow implicit casts that will result in a loss of precision. For example, there should not be an implicit cast from **Double** to **Int32**, but there might be one from **Int32** to **Int64**.
- Do not throw exceptions from implicit casts because it is very difficult for the developer to understand what is happening.
- Provide casts that operate on an entire object. The value that is cast should represent the entire object, not a member of an object. For example, it is not appropriate for a **Button** to cast to a string by returning its caption.
- Do not generate a semantically different value. For example, it is appropriate to convert a **DateTime** or **TimeSpan** into an **Int32**. The **Int32** still represents the time or duration. It does not, however, make sense to convert a file name string such as "c:\mybitmap.gif" into a **Bitmap** object.
- Do not cast values from different domains. Casts operate within a particular domain of values. For example, numbers and strings are different domains. It makes sense that an **Int32** can cast to a **Double**. However, it does not make sense for an **Int32** to cast to a **String**, because they are in different domains.

See Also

[Design Guidelines for Class Library Developers](#)

Common Design Patterns

.NET Framework 1.1

This topic provides guidelines for implementing common design patterns in class libraries.

In This Section

[Implementing Finalize and Dispose to Clean Up Unmanaged Resources](#)

Describes the recommended design pattern to implement in class libraries to clean up unmanaged resources using the **Finalize** and **Dispose** methods.

[Implementing the Equals Method](#)

Describes the guidelines to follow to implement the **Equals** method in class libraries.

[Callback Function Usage](#)

Describes when to use delegates, events, and interfaces to provide callback functionality.

[Timeout Usage](#)

Describes the guidelines for using time-outs in base class libraries to specify the maximum time a caller is willing to wait for completion of a method call.

Related Sections

[Design Guidelines for Class Library Developers](#)

Security in Class Libraries

.NET Framework 1.1

Class library designers must understand [code access security](#) in order to write secure class libraries. When writing a class library, be aware of two security principles: use permissions to help protect objects, and write fully trusted code. The degree to which these principles apply will depend upon the class you are writing. Some classes, such as the [System.IO.FileStream Class](#), represent objects that need protection with permissions. The implementation of these classes is responsible for checking the permissions of callers and allowing only authorized callers to perform operations for which they have permission. The [System.Security Namespace](#) contains classes that can help you perform these checks in the class libraries that you write. Class library code often is fully trusted or at least highly trusted code. Because class library code often accesses protected resources and unmanaged code, any flaws in the code represent a serious threat to the integrity of the entire security system. To help minimize security threats, follow the guidelines described in this topic when writing class library code. For more information, see [Writing Secure Class Libraries](#).

Protecting Objects with Permissions

[Permissions](#) are defined to help protect specific resources. A class library that performs operations on protected resources must be responsible for enforcing this protection. Before acting on any request on a protected resource, such as deleting a file, class library code first must check that the caller (and usually all callers, by means of a stack walk) has the appropriate delete permission for the resource. If the caller has the permission, the action should be allowed to complete. If the caller does not have the permission, the action should not be allowed to complete and a security exception should be raised. Protection is typically implemented in code with either a [declarative](#) or an [imperative](#) check of the appropriate permissions.

It is important that classes protect resources, not only from direct access, but from all possible kinds of exposure. For example, a cached file object is responsible for checking for file read permissions, even if the actual data is retrieved from a cache in memory and no actual file operation occurs. This is because the effect of handing the data to the caller is the same as if the caller had performed an actual read operation.

Fully Trusted Class Library Code

Many class libraries are implemented as fully trusted code that encapsulates platform-specific functionality as managed objects, such as COM or system APIs. Fully trusted code can expose a weakness to the security of the entire system. However, if class libraries are written correctly with respect to security, placing a heavy security burden on a relatively small set of class libraries and the core runtime security allows the larger body of managed code to acquire the security benefits of these core class libraries.

In a common class library security scenario, a fully trusted class exposes a resource that is protected by a permission; the resource is accessed by a native code API. A typical example of this type of resource is a file. The [File class](#) uses a native API to perform file operations, such as a deletion. The following steps are taken to protect the resource.

1. A caller requests the deletion of file `c:\test.txt` by calling the [File.Delete Method](#).
2. The **Delete** method creates a permission object representing the `delete c:\test.txt` permission.
3. The **File** class's code checks all callers on the stack to see if they have been granted the demanded permission; if not, a security exception is raised.
4. The **File** class asserts FullTrust in order to call native code, because its callers might not have this permission.
5. The **File** class uses a native API to perform the file delete operation.
6. The **File** class returns to its caller, and the file delete request is completed successfully.

Precautions for Highly Trusted Code

Code in a trusted class library is granted permissions that are not available to most application code. In addition, an assembly might contain classes that do not need special permissions but are granted these permissions because the assembly contains other classes that do require them. These situations can expose a security weakness to the system. Therefore, you must be take special care when writing highly or fully trusted code.

Design trusted code so that it can be called by any semi-trusted code on the system without exposing security holes. Resources are normally protected by a stack walk of all callers. If a caller has insufficient permissions, attempted access is blocked. However, any time trusted code asserts a permission, the code takes responsibility for checking for required permissions. Normally, an assert should follow a permission check of the caller as described earlier in this topic. In addition, the number of higher permission asserts should be minimized to reduce the risk of unintended exposure.

Fully trusted code is implicitly granted all other permissions. In addition, it is allowed to violate rules of type safety and object usage. Independent of the protection of resources, any aspect of the programmatic interface that might break type safety or allow access to data not normally available to the caller can lead to a security problem.

Performance

Security checks involve checking the stack for the permissions of all callers. Depending upon the depth of the stack, these operations have the potential to be very expensive. If one operation actually consists of a number of actions at a lower level that require security checks, it might greatly improve performance to check caller permissions once and then assert the necessary permission before performing the actions. The assert will stop the stack walk from propagating further up the stack so that the check will stop there and succeed. This technique typically results in a performance improvement if three or more permission checks can be covered at once.

Summary of Class Library Security Issues

- Any class library that uses protected resources must ensure that it does so only within the permissions of its callers.
- Assertion of permissions should be done only when necessary, and should be preceded by the necessary permission checks.
- To improve performance, aggregate operations that will involve security checks and consider the use of assert to limit stack walks without compromising security.
- Be aware of how a semi-trusted malicious caller might potentially use a class to bypass security.
- Do not assume that code will be called only by callers with certain permissions.
- Do not define non-type-safe interfaces that might be used to bypass security elsewhere.
- Do not expose functionality in a class that allows a semi-trusted caller to take advantage of the higher trust of the class.

See Also

[Design Guidelines for Class Library Developers](#) | [Writing Secure Class Libraries](#) | [Code Access Security](#) | Security and Culture-Aware String Operations

Threading Design Guidelines

.NET Framework 1.1

The following rules outline the design guidelines for implementing threading:

- Avoid providing static methods that alter static state. In common server scenarios, static state is shared across requests, which means multiple threads can execute that code at the same time. This opens up the possibility for threading bugs. Consider using a design pattern that encapsulates data into instances that are not shared across requests.
- Static state must be thread safe.
- Instance state does not need to be thread safe. By default, class libraries should not be thread safe. Adding locks to create thread-safe code decreases performance, increases lock contention, and creates the possibility for deadlock bugs to occur. In common application models, only one thread at a time executes user code, which minimizes the need for thread safety. For this reason, the .NET Framework class libraries are not thread safe by default. In cases where you want to provide a thread-safe version, provide a static **Synchronized** method that returns a thread-safe instance of a type. For an example, see the [System.Collections.ArrayList.Synchronized Method](#) and the [System.Collections.ArrayList.IsSynchronized Method](#).
- Design your library with consideration for the stress of running in a server scenario. Avoid taking locks whenever possible.
- Be aware of method calls in locked sections. Deadlocks can result when a static method in class A calls static methods in class B and vice versa. If A and B both synchronize their static methods, this will cause a deadlock. You might discover this deadlock only under heavy threading stress.
- Performance issues can result when a static method in class A calls a static method in class A. If these methods are not factored correctly, performance will suffer because there will be a large amount of redundant synchronization. Excessive use of fine-grained synchronization might negatively impact performance. In addition, it might have a significant negative impact on scalability.
- Be aware of issues with the **lock** statement (**SyncLock** in Visual Basic). It is tempting to use the **lock** statement to solve all threading problems. However, the [System.Threading.Interlocked Class](#) is superior for updates that must be atomic. It executes a single **lock** prefix if there is no contention. In a code review, you should watch out for instances like the one shown in the following example.

VB

```
SyncLock Me
    myField += 1
End SyncLock
[C#]
lock(this)
{
    myField++;
}
```

If you replace the previous example with the following one, you will improve performance.

VB

```
System.Threading.Interlocked.Increment(myField)
[C#]
System.Threading.Interlocked.Increment(myField);
```

Another example is to update an object type variable only if it is null (**Nothing** in Visual Basic). You can use the following code to update the variable and make the code thread safe.

VB

```
If x Is Nothing Then
    SyncLock Me
        If x Is Nothing Then
            x = y
        End If
    End SyncLock
End If
[C#]
if (x == null)
{
    lock (this)
    {
        if (x == null)
        {
            x = y;
        }
    }
}
```

You can improve the performance of the previous sample by replacing it with the following code.

VB

```
System.Threading.Interlocked.CompareExchange(x, y, Nothing)
[C#]
System.Threading.Interlocked.CompareExchange(ref x, y, null);
```

- Avoid the need for synchronization if possible. For high traffic pathways, it is best to avoid synchronization. Sometimes the algorithm can be adjusted to tolerate race conditions rather than eliminate them.

See Also

[Design Guidelines for Class Library Developers](#) | [Visual Basic Language Changes](#) | [System.Threading Namespace](#)

Guidelines for Asynchronous Programming

.NET Framework 1.1

Asynchronous programming is a feature supported by many areas of the common language runtime, such as [Remoting](#), [ASP.NET](#), and Windows Forms. Asynchronous programming is a core concept in the .NET Framework. This topic introduces the design pattern for asynchronous programming.

The philosophy behind these guidelines is as follows:

- The client should decide whether a particular call should be asynchronous.
- It is not necessary for a server to do additional programming in order to support its clients' asynchronous behavior. The runtime should be able to manage the difference between the client and server views. As a result, the situation where the server has to implement **IDispatch** and do a large amount of work to support dynamic invocation by clients is avoided.
- The server can choose to explicitly support asynchronous behavior either because it can implement asynchronous behavior more efficiently than a general architecture, or because it wants to support only asynchronous behavior by its clients. It is recommended that such servers follow the design pattern outlined in this document for exposing asynchronous operations.
- Type safety must be enforced.
- The runtime provides the necessary services to support the asynchronous programming model. These services include the following:
 - Synchronization primitives, such as critical sections and [ReaderWriterLock](#) instances.
 - Synchronization constructs such as containers that support the **WaitForMultipleObjects** method.
 - Thread pools.
 - Exposure to the underlying infrastructure, such as [Message](#) and [ThreadPool](#) objects.

See Also

[Design Guidelines for Class Library Developers](#) | [Asynchronous Programming Design Pattern](#)

Asynchronous Programming Design Pattern

.NET Framework 1.1

The following code example demonstrates a server class that factorizes a number.

VB

```
Public Class PrimeFactorizer
    Public Function Factorize(factorizableNum As Long, ByRef primefactor1
        As Long, ByRef primefactor2 As Long) As Boolean
        primefactor1 = 1
        primefactor2 = factorizableNum

        ' Factorize using a low-tech approach.
        Dim i As Integer
        For i = 2 To factorizableNum - 1
            If 0 = factorizableNum Mod i Then
                primefactor1 = i
                primefactor2 = factorizableNum / i
                Exit For
            End If
        Next i
        If 1 = primefactor1 Then
            Return False
        Else
            Return True
        End If
    End Function
End Class
[C#]
public class PrimeFactorizer
{
    public bool Factorize(long factorizableNum,
        ref long primefactor1,
        ref long primefactor2)
    {
        primefactor1 = 1;
        primefactor2 = factorizableNum;

        // Factorize using a low-tech approach.
        for (int i=2; i<factorizableNum; i++)
        {
            if (0 == (factorizableNum % i))
            {
                primefactor1 = i;
                primefactor2 = factorizableNum / i;
                break;
            }
        }
        if (1 == primefactor1 )
            return false;
        else
            return true ;
    }
}
```

The following code example shows a client defining a pattern for asynchronously invoking the Factorize method from the PrimeFactorizer class in the previous example.

VB

```
' Define the delegate.
Delegate Function FactorizingAsyncDelegate(factorizableNum As Long, ByRef
    primefactor1 As Long, ByRef primefactor2 As Long)
End Sub
' Create an instance of the Factorizer.
Dim pf As New PrimeFactorizer()

' Create a delegate on the Factorize method on the Factorizer.
Dim fd As New FactorizingDelegate(pf.Factorize)
[C#]
// Define the delegate.
public delegate bool FactorizingAsyncDelegate(long factorizableNum,
    ref long primefactor1,
    ref long primefactor2);

// Create an instance of the Factorizer.
PrimeFactorizer pf = new PrimeFactorizer();

// Create a delegate on the Factorize method on the Factorizer.
FactorizingDelegate fd = new FactorizingDelegate(pf.Factorize);
```

The compiler will emit the following **FactorizingAsyncDelegate** class after parsing its definition in the first line of the previous example. It will generate the **BeginInvoke** and **EndInvoke** methods.

VB

```
Public Class FactorizingAsyncDelegate
    Inherits Delegate

    Public Function Invoke(factorizableNum As Long, ByRef primefactor1 As
        Long, ByRef primefactor2 As Long) As Boolean
    End Function

    ' Supplied by the compiler.
    Public Function BeginInvoke(factorizableNum As Long, ByRef primefactor1
        As Long, ByRef primefactor2 As Long, cb As AsyncCallback,
        AsyncState As Object) As
        IAsyncResult
    End Function

    ' Supplied by the compiler.
    Public Function EndInvoke(ByRef primefactor1 As Long, ByRef
        primefactor2 As Long, ar As IAsyncResult) As Boolean
    End Function
End Class
```

```
[C#]
public class FactorizingAsyncDelegate : Delegate
{
    public bool Invoke(ulong factorizableNum,
        ref ulong primefactor1, ref ulong primefactor2);

    // Supplied by the compiler.
    public IAsyncResult BeginInvoke(ulong factorizableNum,
        ref unsigned long primefactor1,
        ref unsigned long primefactor2, AsyncCallback cb,
        Object AsyncState);

    // Supplied by the compiler.
    public bool EndInvoke(ref ulong primefactor1,
        ref ulong primefactor2, IAsyncResult ar);
}
```

The interface used as the delegate parameter in the following code example is defined in the .NET Framework Class Library. For more information, see [IAsyncResult Interface](#).

VB

```
Delegate Function AsyncCallback(ar As IAsyncResult)

' Returns true if the asynchronous operation has been completed.
Public Interface IAsyncResult
    ' Handle to block on for the results.
    ReadOnly Property IsCompleted() As Boolean
    ' Get accessor implementation goes here.
    End Property

    ' Caller can use this to wait until operation is complete.
    ReadOnly Property AsyncWaitHandle() As WaitHandle
    ' Get accessor implementation goes here.
    End Property

    ' The delegate object for which the async call was invoked.
    ReadOnly Property AsyncObject() As [Object]
    ' Get accessor implementation goes here.
    End Property

    ' The state object passed in through BeginInvoke.
    ReadOnly Property AsyncState() As [Object]
    ' Get accessor implementation goes here.
    End Property

    ' Returns true if the call completed synchronously.
    ReadOnly Property CompletedSynchronously() As Boolean
    ' Get accessor implementation goes here.
    End Property
End Interface
```

```
[C#]
public delegate AsyncCallback (IAsyncResult ar);

public interface IAsyncResult
{
    // Returns true if the asynchronous operation has completed.
    bool IsCompleted { get; }

    // Caller can use this to wait until operation is complete.
    WaitHandle AsyncWaitHandle { get; }

    // The delegate object for which the async call was invoked.
    Object AsyncObject { get; }
}
```



```

// The state object passed in through BeginInvoke.
Object AsyncState { get; }

// Returns true if the call completed synchronously.
bool CompletedSynchronously { get; }
}

```

Note that the object that implements the [IAsyncResult Interface](#) must be a waitable object and its underlying synchronization primitive should be signaled after the call is canceled or completed. This enables the client to wait for the call to complete instead of polling. The runtime supplies a number of waitable objects that mirror Win32 synchronization primitives, such as [ManualResetEvent](#), [AutoResetEvent](#) and [Mutex](#). It also supplies methods that support waiting for such synchronization objects to become signaled with "any" or "all" semantics. Such methods are context-aware to avoid deadlocks.

The **Cancel** method is a request to cancel processing of the method after the desired time-out period has expired. Note that it is only a request by the client and the server is recommended to honor it. Further, the client should not assume that the server has stopped processing the request completely after receiving notification that the method has been canceled. In other words, the client is recommended to not destroy resources such as file objects, as the server might be actively using them. The **IsCanceled** property will be set to **true** if the call was canceled and the **IsCompleted** property will be set to **true** after the server has completed processing of the call. After the server sets the **IsCompleted** property to **true**, the server cannot use any client-supplied resources outside of the agreed-upon sharing semantics. Thus, it is safe for the client to destroy the resources after the **IsCompleted** property returns **true**.

The **Server** property returns the server object that provided the **IAsyncResult**.

The following code example demonstrates the client-side programming model for invoking the **Factorize** method asynchronously.

C#

```

public class ProcessFactorizeNumber
{
    private long _ulNumber;

    public ProcessFactorizeNumber(long number)
    {
        _ulNumber = number;
    }

    [OneWayAttribute()]
    public void FactorizedResults(IAsyncResult ar)
    {
        long factor1=0, factor2=0;

        // Extract the delegate from the AsyncResult.
        FactorizingAsyncDelegate fd =
            (FactorizingAsyncDelegate)((AsyncResult)ar).AsyncDelegate;
        // Obtain the result.
        fd.EndInvoke(ref factor1, ref factor2, ar);

        // Output the results.
        Console.WriteLine("On CallBack: Factors of {0} : {1} {2}",
            _ulNumber, factor1, factor2);
    }
}

// Async Variation 1.
// The ProcessFactorizeNumber.FactorizedResults callback function
// is called when the call completes.
public void FactorizeNumber1()
{
    // Client code.
    PrimeFactorizer pf = new PrimeFactorizer();
    FactorizingAsyncDelegate fd = new FactorizingAsyncDelegate (pf.Factorize);

    long factorizableNum = 1000589023, temp=0;

    // Create an instance of the class that
    // will be called when the call completes.
    ProcessFactorizeNumber fc =
        new ProcessFactorizeNumber(factorizableNum);
    // Define the AsyncCallback delegate.
    AsyncCallbackDelegate cb = new AsyncCallbackDelegate (fc.FactorizedResults);
    // Any object can be the state object.
    Object state = new Object();

    // Asynchronously invoke the Factorize method on pf.
    // Note: If you have pure out parameters, you do not need the
    // temp variable.
    IAsyncResult ar = fd.BeginInvoke(factorizableNum, ref temp, ref temp,
        cb, state);

    // Proceed to do other useful work.

    // Async Variation 2.
    // Waits for the result.
    // Asynchronously invoke the Factorize method on pf.
    // Note: If you have pure out parameters, you do not need
    // the temp variable.
    public void FactorizeNumber2()
    {

```

```

// Client code.
PrimeFactorizer pf = new PrimeFactorizer();
FactorizingAsyncDelegate fd = new FactorizingAsyncDelegate (pf.Factorize);

long factorizableNum = 1000589023, temp=0;
// Create an instance of the class
// to be called when the call completes.
ProcessFactorizedNumber fc =
    new ProcessFactorizedNumber(factorizableNum);

// Define the AsyncCallback delegate.
AsyncCallback cb = new AsyncCallback(fc.FactorizedResults);

// Any object can be the state object.
Object state = new Object();

// Asynchronously invoke the Factorize method on pf.
IAsyncResult ar = fd.BeginInvoke(factorizableNum, ref temp, ref temp,
    null, null);

ar.AsyncWaitHandle.WaitOne(10000, false);

if(ar.IsCompleted)
{
    int factor1=0, factor2=0;

    // Obtain the result.
    fd.EndInvoke(ref factor1, ref factor2, ar);

    // Output the results.
    Console.WriteLine("Sequential : Factors of {0} : {1} {2}",
        factorizableNum, factor1, factor2);
}
}

```

Note that if `FactorizeCallback` is a context-bound class that requires synchronized or thread-affinity context, the callback function is dispatched through the context dispatcher infrastructure. In other words, the callback function itself might execute asynchronously with respect to its caller for such contexts. These are the semantics of the one-way qualifier on method signatures. Any such method call might execute synchronously or asynchronously with respect to caller, and the caller cannot make any assumptions about completion of such a call when execution control returns to it.

Also, calling **EndInvoke** before the asynchronous operation is complete will block the caller. Calling it a second time with the same `AsyncResult` is undefined.

Summary of Asynchronous Programming Design Pattern

The server splits an asynchronous operation into its two logical parts: the part that takes input from the client and starts the asynchronous operation, and the part that supplies the results of the asynchronous operation to the client. In addition to the input needed for the asynchronous operation, the first part also takes an **AsyncCallbackDelegate** object to be called when the asynchronous operation is completed. The first part returns a waitable object that implements the **IAsyncResult** interface used by the client to determine the status of the asynchronous operation. The server typically also uses the waitable object it returned to the client to maintain any state associated with asynchronous operation. The client uses the second part to obtain the results of the asynchronous operation by supplying the waitable object.

When initiating asynchronous operations, the client can either supply the callback function delegate or not supply it.

The following options are available to the client for completing asynchronous operations:

- Poll the returned **IAsyncResult** object for completion.
- Attempt to complete the operation prematurely, thereby blocking until the operation completes.
- Wait on the **IAsyncResult** object. The difference between this and the previous option is that the client can use time-outs to periodically take back control.
- Complete the operation inside the callback function routine.

One scenario in which both synchronous and asynchronous read and write methods are desirable is the use of file input/output. The following example illustrates the design pattern by showing how the **File** object implements read and write operations.

VB

```

Public Class File
    ' Other methods for this class go here.

    ' Synchronous read method.
    Function Read(buffer() As [Byte], NumToRead As Long) As Long

    ' Asynchronous read method.
    Function BeginRead(buffer() As [Byte], NumToRead As Long, cb As
        AsyncCallbackDelegate) As IAsyncResult
    Function EndRead(ar As IAsyncResult) As Long

    ' Synchronous write method.
    Function Write(buffer() As [Byte], NumToWrite As Long) As Long

    ' Asynchronous write method.
    Function BeginWrite(buffer() As [Byte], NumToWrite As Long, cb As
        AsyncCallbackDelegate) As IAsyncResult
    Function EndWrite(ar As IAsyncResult) As Long
End Class
[C#]
public class File
{
    // Other methods for this class go here.

```

```
// Synchronous read method.  
long Read(Byte[] buffer, long NumToRead);  
  
// Asynchronous read method.  
IAsyncResult BeginRead(Byte[] buffer, long NumToRead,  
    AsyncCallback cb);  
long EndRead(IAsyncResult ar);  
  
// Synchronous write method.  
long Write(Byte[] buffer, long NumToWrite);  
  
// Asynchronous write method.  
IAsyncResult BeginWrite(Byte[] buffer, long NumToWrite,  
    AsyncCallback cb);  
  
long EndWrite(IAsyncResult ar);  
}
```

The client cannot easily associate state with a given asynchronous operation without defining a new callback function delegate for each operation. This can be fixed by making `Begin` methods, such as `BeginWrite`, take an extra object parameter that represents the state and which is captured in the **`IAsyncResult`**.

See Also

[Design Guidelines for Class Library Developers](#) | [Asynchronous Execution](#) | [IAsyncResult Interface](#)

Framework Design Guidelines

.NET Framework 4.5

This section provides guidelines for designing libraries that extend and interact with the .NET Framework. The goal is to help library designers ensure API consistency and ease of use by providing a unified programming model that is independent of the programming language used for development. We recommend that you follow these design guidelines when developing classes and components that extend the .NET Framework. Inconsistent library design adversely affects developer productivity and discourages adoption.

The guidelines are organized as simple recommendations prefixed with the terms **Do**, **Consider**, **Avoid**, and **Do not**. These guidelines are intended to help class library designers understand the trade-offs between different solutions. There might be situations where good library design requires that you violate these design guidelines. Such cases should be rare, and it is important that you have a clear and compelling reason for your decision.

These guidelines are excerpted from the book *Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries, 2nd Edition*, by Krzysztof Cwalina and Brad Abrams.

▲ In This Section

[Naming Guidelines](#)

Provides guidelines for naming assemblies, namespaces, types, and members in class libraries.

[Type Design Guidelines](#)

Provides guidelines for using static and abstract classes, interfaces, enumerations, structures, and other types.

[Member Design Guidelines](#)

Provides guidelines for designing and using properties, methods, constructors, fields, events, operators, and parameters.

[Designing for Extensibility](#)

Discusses extensibility mechanisms such as subclassing, using events, virtual members, and callbacks, and explains how to choose the mechanisms that best meet your framework's requirements.

[Design Guidelines for Exceptions](#)

Describes design guidelines for designing, throwing, and catching exceptions.

[Usage Guidelines](#)

Describes guidelines for using common types such as arrays, attributes, and collections, supporting serialization, and overloading equality operators.

[Common Design Patterns](#)

Provides guidelines for choosing and implementing dependency properties and the dispose pattern.

Portions © 2005, 2009 Microsoft Corporation. All rights reserved.

Reprinted by permission of Pearson Education, Inc. from [Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries, 2nd Edition](#) by Krzysztof Cwalina and Brad Abrams, published Oct 22, 2008 by Addison-Wesley Professional as part of the Microsoft Windows Development Series.

▲ See Also

Concepts

[Overview of the .NET Framework](#)

[Roadmap for the .NET Framework](#)

Other Resources

[.NET Framework Development Guide](#)

Naming Guidelines

.NET Framework 4.5

Following a consistent set of naming conventions in the development of a framework can be a major contribution to the framework's usability. It allows the framework to be used by many developers on widely separated projects. Beyond consistency of form, names of framework elements must be easily understood and must convey the function of each element.

The goal of this chapter is to provide a consistent set of naming conventions that results in names that make immediate sense to developers.

Although adopting these naming conventions as general code development guidelines would result in more consistent naming throughout your code, you are required only to apply them to APIs that are publicly exposed (public or protected types and members, and explicitly implemented interfaces).

▲ In This Section

- [Capitalization Conventions](#)
- [General Naming Conventions](#)
- [Names of Assemblies and DLLs](#)
- [Names of Namespaces](#)
- [Names of Classes, Structs, and Interfaces](#)
- [Names of Type Members](#)
- [Naming Parameters](#)
- [Naming Resources](#)

Portions © 2005, 2009 Microsoft Corporation. All rights reserved.

Reprinted by permission of Pearson Education, Inc. from [Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries, 2nd Edition](#) by Krzysztof Cwalina and Brad Abrams, published Oct 22, 2008 by Addison-Wesley Professional as part of the Microsoft Windows Development Series.

▲ See Also

- [Other Resources](#)
- [Framework Design Guidelines](#)

Capitalization Conventions

.NET Framework 4.5

The guidelines in this chapter lay out a simple method for using case that, when applied consistently, make identifiers for types, members, and parameters easy to read.

Capitalization Rules for Identifiers

To differentiate words in an identifier, capitalize the first letter of each word in the identifier. Do not use underscores to differentiate words, or for that matter, anywhere in identifiers. There are two appropriate ways to capitalize identifiers, depending on the use of the identifier:

- PascalCasing
- camelCasing

The PascalCasing convention, used for all identifiers except parameter names, capitalizes the first character of each word (including acronyms over two letters in length), as shown in the following examples:

PropertyDescriptor
Html Tag

A special case is made for two-letter acronyms in which both letters are capitalized, as shown in the following identifier:

IOStream

The camelCasing convention, used only for parameter names, capitalizes the first character of each word except the first word, as shown in the following examples. As the example also shows, two-letter acronyms that begin a camel-cased identifier are both lowercase.

propertyDescriptor
ioStream
html Tag

✓ **DO** use PascalCasing for all public member, type, and namespace names consisting of multiple words.

✓ **DO** use camelCasing for parameter names.

The following table describes the capitalization rules for different types of identifiers.

Identifier	Casing	Example
Namespace	Pascal	namespace System.Security { ... }
Type	Pascal	public class StreamReader { ... }
Interface	Pascal	public interface IEnumerable { ... }
Method	Pascal	public class Object { public virtual string ToString(); }
Property	Pascal	public class String { public int Length { get; } }
Event	Pascal	public class Process { public event EventHandler Exited; }
Field	Pascal	public class MessageQueue { public static readonly TimeSpan InfiniteTimeout; } public struct UInt32 { public const Min = 0; }
Enum value	Pascal	public enum FileMode { Append, ... }
Parameter	Camel	public class Convert { public static int ToInt32(string value); }

Capitalizing Compound Words and Common Terms

Most compound terms are treated as single words for purposes of capitalization.

X DO NOT capitalize each word in so-called closed-form compound words.

These are compound words written as a single word, such as endpoint. For the purpose of casing guidelines, treat a closed-form compound word as a single word. Use a current dictionary to determine if a compound word is written in closed form.

Pascal	Camel	Not
Bi tFl ag	bi tFl ag	Bi tfl ag
Cal l back	cal l back	Cal l Back
Cancel ed	cancel ed	Cancel l ed
DoNot	doNot	Don' t
Emai l	emai l	EMai l
Endpoi nt	endpoi nt	EndPoi nt
Fi l eName	fi l eName	Fi l ename
Grid l i ne	grid l i ne	GridLi ne
Hashtabl e	hashtabl e	HashTabl e
Id	i d	ID
Indexes	i ndexes	Indl ces
LogOff	l ogOff	LogOut
LogOn	l ogOn	Logl n
Metadata	metadata	MetaData, metaData
Mul ti panel	mul ti panel	Mul ti Panel
Mul ti vi ew	mul ti vi ew	Mul ti Vi ew
Namespace	namespace	NameSpace
Ok	ok	OK
Pi	pi	PI
Pl acehol der	pl acehol der	Pl aceHol der
Si gnI n	si gnI n	Si gnOn
Si gnOut	si gnOut	Si gnOff
UserName	userName	Username
Whi teSpace	whi teSpace	Whi tespace
Wri tabl e	wri tabl e	Wri teabl e

Case Sensitivity

Languages that can run on the CLR are not required to support case-sensitivity, although some do. Even if your language supports it, other languages that might access your framework do not. Any APIs that are externally accessible, therefore, cannot rely on case alone to distinguish between two names in the same context.

X DO NOT assume that all programming languages are case sensitive. They are not. Names cannot differ by case alone.

Portions © 2005, 2009 Microsoft Corporation. All rights reserved.

Reprinted by permission of Pearson Education, Inc. from [Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries, 2nd Edition](#) by Krzysztof Cwalina and Brad Abrams, published Oct 22, 2008 by Addison-Wesley Professional as part of the Microsoft Windows Development Series.

▲ See Also

Other Resources

[Framework Design Guidelines](#)

[Naming Guidelines](#)

General Naming Conventions

.NET Framework 4.5

This section describes general naming conventions that relate to word choice, guidelines on using abbreviations and acronyms, and recommendations on how to avoid using language-specific names.

Word Choice

✓ **DO** choose easily readable identifier names.

For example, a property named `HorizontalAlignment` is more English-readable than `AlignmentHorizontal`.

✓ **DO** favor readability over brevity.

The property name `CanScrollHorizontally` is better than `ScrollableX` (an obscure reference to the X-axis).

✗ **DO NOT** use underscores, hyphens, or any other nonalphanumeric characters.

✗ **DO NOT** use Hungarian notation.

✗ **AVOID** using identifiers that conflict with keywords of widely used programming languages.

According to Rule 4 of the Common Language Specification (CLS), all compliant languages must provide a mechanism that allows access to named items that use a keyword of that language as an identifier. C#, for example, uses the `@` sign as an escape mechanism in this case. However, it is still a good idea to avoid common keywords because it is much more difficult to use a method with the escape sequence than one without it.

Using Abbreviations and Acronyms

✗ **DO NOT** use abbreviations or contractions as part of identifier names.

For example, use `GetWindow` rather than `GetWin`.

✗ **DO NOT** use any acronyms that are not widely accepted, and even if they are, only when necessary.

Avoiding Language-Specific Names

✓ **DO** use semantically interesting names rather than language-specific keywords for type names.

For example, `GetLength` is a better name than `GetInt`.

✓ **DO** use a generic CLR type name, rather than a language-specific name, in the rare cases when an identifier has no semantic meaning beyond its type.

For example, a method converting to `Int64` should be named `ToInt64`, not `ToLong` (because `Int64` is a CLR name for the C#-specific alias `long`). The following table presents several base data types using the CLR type names (as well as the corresponding type names for C#, Visual Basic, and C++).

C#	Visual Basic	C++	CLR
sbyte	SByte	char	SByte
byte	Byte	unsigned char	Byte
short	Short	short	Int16
ushort	UInt16	unsigned short	UInt16
int	Integer	int	Int32
uint	UInt32	unsigned int	UInt32
long	Long	_int64	Int64
ulong	UInt64	unsigned _int64	UInt64
float	Single	float	Single
double	Double	double	Double
bool	Boolean	bool	Boolean
char	Char	wchar_t	Char
string	String	String	String

object	Object	Object	Object
--------	--------	--------	--------

✓ **DO** use a common name, such as *value* or *item*, rather than repeating the type name, in the rare cases when an identifier has no semantic meaning and the type of the parameter is not important.

Naming New Versions of Existing APIs

✓ **DO** use a name similar to the old API when creating new versions of an existing API.

This helps to highlight the relationship between the APIs.

✓ **DO** prefer adding a suffix rather than a prefix to indicate a new version of an existing API.

This will assist discovery when browsing documentation, or using Intellisense. The old version of the API will be organized close to the new APIs, because most browsers and Intellisense show identifiers in alphabetical order.

✓ **CONSIDER** using a brand new, but meaningful identifier, instead of adding a suffix or a prefix.

✓ **DO** use a numeric suffix to indicate a new version of an existing API, particularly if the existing name of the API is the only name that makes sense (i.e., if it is an industry standard) and if adding any meaningful suffix (or changing the name) is not an appropriate option.

✗ **DO NOT** use the "Ex" (or a similar) suffix for an identifier to distinguish it from an earlier version of the same API.

✓ **DO** use the "64" suffix when introducing versions of APIs that operate on a 64-bit integer (a long integer) instead of a 32-bit integer. You only need to take this approach when the existing 32-bit API exists; don't do it for brand new APIs with only a 64-bit version.

Portions © 2005, 2009 Microsoft Corporation. All rights reserved.

Reprinted by permission of Pearson Education, Inc. from [Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries, 2nd Edition](#) by Krzysztof Cwalina and Brad Abrams, published Oct 22, 2008 by Addison-Wesley Professional as part of the Microsoft Windows Development Series.

See Also

Other Resources
[Framework Design Guidelines](#)
[Naming Guidelines](#)

Names of Assemblies and DLLs

.NET Framework 4.5

An assembly is the unit of deployment and identity for managed code programs. Although assemblies can span one or more files, typically an assembly maps one-to-one with a DLL. Therefore, this section describes only DLL naming conventions, which then can be mapped to assembly naming conventions.

✓ **DO** choose names for your assembly DLLs that suggest large chunks of functionality, such as System.Data.

Assembly and DLL names don't have to correspond to namespace names, but it is reasonable to follow the namespace name when naming assemblies. A good rule of thumb is to name the DLL based on the common prefix of the assemblies contained in the assembly. For example, an assembly with two namespaces, `MyCompany.MyTechnology.FirstFeature` and `MyCompany.MyTechnology.SecondFeature`, could be called `MyCompany.MyTechnology.dll`.

✓ **CONSIDER** naming DLLs according to the following pattern:

`<Company>. <Component>. dll`

where `<Component>` contains one or more dot-separated clauses. For example:

`Li tware. Control s. dll`.

Portions © 2005, 2009 Microsoft Corporation. All rights reserved.

Reprinted by permission of Pearson Education, Inc. from [Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries, 2nd Edition](#) by Krzysztof Cwalina and Brad Abrams, published Oct 22, 2008 by Addison-Wesley Professional as part of the Microsoft Windows Development Series.

See Also

Other Resources

[Framework Design Guidelines](#)

[Naming Guidelines](#)

Names of Namespaces

.NET Framework 4.5

As with other naming guidelines, the goal when naming namespaces is creating sufficient clarity for the programmer using the framework to immediately know what the content of the namespace is likely to be. The following template specifies the general rule for naming namespaces:

`<Company>. (<Product>|<Technology>)[. <Feature>][. <Subnamespace>]`

The following are examples:

`Fabrikam.Math`
`Librium.Security`

✓ **DO** prefix namespace names with a company name to prevent namespaces from different companies from having the same name.

✓ **DO** use a stable, version-independent product name at the second level of a namespace name.

✗ **DO NOT** use organizational hierarchies as the basis for names in namespace hierarchies, because group names within corporations tend to be short-lived. Organize the hierarchy of namespaces around groups of related technologies.

✓ **DO** use PascalCasing, and separate namespace components with periods (e.g., `Microsoft.Office.PowerPoint`). If your brand employs nontraditional casing, you should follow the casing defined by your brand, even if it deviates from normal namespace casing.

✓ **CONSIDER** using plural namespace names where appropriate.

For example, use `System.Collections` instead of `System.Collecti on`. Brand names and acronyms are exceptions to this rule, however. For example, use `System.IO` instead of `System.IOs`.

✗ **DO NOT** use the same name for a namespace and a type in that namespace.

For example, do not use `Debug` as a namespace name and then also provide a class named `Debug` in the same namespace. Several compilers require such types to be fully qualified.

Namespaces and Type Name Conflicts

✗ **DO NOT** introduce generic type names such as `Element`, `Node`, `Log`, and `Message`.

There is a very high probability that doing so will lead to type name conflicts in common scenarios. You should qualify the generic type names (`FormElement`, `XmlNode`, `EventLog`, `SoapMessage`).

There are specific guidelines for avoiding type name conflicts for different categories of namespaces.

- **Application model namespaces**

Namespaces belonging to a single application model are very often used together, but they are almost never used with namespaces of other application models. For example, the `System.Windows.Forms` namespace is very rarely used together with the `System.Web.UI` namespace. The following is a list of well-known application model namespace groups:

`System.Windows*`
`System.Web.*`

✗ **DO NOT** give the same name to types in namespaces within a single application model.

For example, do not add a type named `Page` to the `System.Web.UI.Adapters` namespace, because the `System.Web.UI` namespace already contains a type named `Page`.

- **Infrastructure namespaces**

This group contains namespaces that are rarely imported during development of common applications. For example, `Design` namespaces are mainly used when developing programming tools. Avoiding conflicts with types in these namespaces is not critical.

- **Core namespaces**

Core namespaces include all `System` namespaces, excluding namespaces of the application models and the Infrastructure namespaces. Core namespaces include, among others, `System`, `System.IO`, `System.Xml`, and `System.Net`.

✗ **DO NOT** give types names that would conflict with any type in the Core namespaces.

For example, never use `Stream` as a type name. It would conflict with `System.IO.Stream`, a very commonly used type.

- **Technology namespace groups**

This category includes all namespaces with the same first two namespace nodes (`<Company>.<Technology>*`), such as `Microsoft.Build.Utilities` and `Microsoft.Build.Tasks`. It is important that types belonging to a single technology do not conflict with each other.

✗ **DO NOT** assign type names that would conflict with other types within a single technology.

✗ **DO NOT** introduce type name conflicts between types in technology namespaces and an application model namespace (unless the technology is not intended to be used with the application model).

Portions © 2005, 2009 Microsoft Corporation. All rights reserved.

Reprinted by permission of Pearson Education, Inc. from [Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries, 2nd Edition](#) by Krzysztof Cwalina and Brad Abrams, published Oct 22, 2008 by Addison-Wesley Professional as part of the Microsoft Windows Development Series.

▲ See Also

Other Resources

[Framework Design Guidelines](#)

[Naming Guidelines](#)

Names of Classes, Structs, and Interfaces

.NET Framework 4.5

The naming guidelines that follow apply to general type naming.

✓ **DO** name classes and structs with nouns or noun phrases, using PascalCasing.

This distinguishes type names from methods, which are named with verb phrases.

✓ **DO** name interfaces with adjective phrases, or occasionally with nouns or noun phrases.

Nouns and noun phrases should be used rarely and they might indicate that the type should be an abstract class, and not an interface.

✗ **DO NOT** give class names a prefix (e.g., "C").

✓ **CONSIDER** ending the name of derived classes with the name of the base class.

This is very readable and explains the relationship clearly. Some examples of this in code are: `ArgumentOutOfRangeException`, which is a kind of `Exception`, and `SerializableAttribute`, which is a kind of `Attribute`. However, it is important to use reasonable judgment in applying this guideline; for example, the `Button` class is a kind of `Control` event, although `Control` doesn't appear in its name.

✓ **DO** prefix interface names with the letter I, to indicate that the type is an interface.

For example, `IComponent` (descriptive noun), `ICustomAttributeProvider` (noun phrase), and `IPersistable` (adjective) are appropriate interface names. As with other type names, avoid abbreviations.

✓ **DO** ensure that the names differ only by the "I" prefix on the interface name when you are defining a class–interface pair where the class is a standard implementation of the interface.

Names of Generic Type Parameters

Generics were added to .NET Framework 2.0. The feature introduced a new kind of identifier called *type parameter*.

✓ **DO** name generic type parameters with descriptive names unless a single-letter name is completely self-explanatory and a descriptive name would not add value.

✓ **CONSIDER** using `T` as the type parameter name for types with one single-letter type parameter.

```
public int IComparer<T> { ... }
public delegate bool Predicate<T>(T item);
public struct Nullable<T> where T:struct { ... }
```

✓ **DO** prefix descriptive type parameter names with `T`.

```
public interface ISessionChannel<TSession> where TSession : ISession{
    TSession Session { get; }
}
```

✓ **CONSIDER** indicating constraints placed on a type parameter in the name of the parameter.

For example, a parameter constrained to `ISession` might be called `TSession`.

Names of Common Types

✓ **DO** follow the guidelines described in the following table when naming types derived from or implementing certain .NET Framework types.

Base Type	Derived/Implementing Type Guideline
<code>System.Attribute</code>	✓ DO add the suffix "Attribute" to names of custom attribute classes. add the suffix "Attribute" to names of custom attribute classes.
<code>System.Delegate</code>	✓ DO add the suffix "EventHandler" to names of delegates that are used in events. ✓ DO add the suffix "Callback" to names of delegates other than those used as event handlers. ✗ DO NOT add the suffix "Delegate" to a delegate.
<code>System.EventArgs</code>	✓ DO add the suffix "EventArgs."
<code>System.Enum</code>	✗ DO NOT derive from this class; use the keyword supported by your language instead; for example, in C#, use the enum keyword. ✗ DO NOT add the suffix "Enum" or "Flag."
<code>System.Exception</code>	✓ DO add the suffix "Exception."

<code>IDictionary</code> <code>IDictionary<TKey, TValue></code>	✓ DO add the suffix "Dictionary." Note that <code>IDictionary</code> is a specific type of collection, but this guideline takes precedence over the more general collections guideline that follows.
<code>IEnumerable</code> <code>ICollection</code> <code> IList</code> <code>IEnumerable<T></code> <code>ICollection<T></code> <code> IList<T></code>	✓ DO add the suffix "Collection."
<code>System.IO.Stream</code>	✓ DO add the suffix "Stream."
<code>CodeAccessPermission</code> <code>Permission</code>	✓ DO add the suffix "Permission."

Naming Enumerations

Names of enumeration types (also called enums) in general should follow the standard type-naming rules (PascalCasing, etc.). However, there are additional guidelines that apply specifically to enums.

- ✓ **DO** use a singular type name for an enumeration unless its values are bit fields.
- ✓ **DO** use a plural type name for an enumeration with bit fields as values, also called flags enum.
- X **DO NOT** use an "Enum" suffix in enum type names.
- X **DO NOT** use "Flag" or "Flags" suffixes in enum type names.
- X **DO NOT** use a prefix on enumeration value names (e.g., "ad" for ADO enums, "rtf" for rich text enums, etc.).

Portions © 2005, 2009 Microsoft Corporation. All rights reserved.

Reprinted by permission of Pearson Education, Inc. from [Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries, 2nd Edition](#) by Krzysztof Cwalina and Brad Abrams, published Oct 22, 2008 by Addison-Wesley Professional as part of the Microsoft Windows Development Series.

See Also

Other Resources
[Framework Design Guidelines](#)
[Naming Guidelines](#)

Names of Type Members

.NET Framework 4.5

Types are made of members: methods, properties, events, constructors, and fields. The following sections describe guidelines for naming type members.

Names of Methods

Because methods are the means of taking action, the design guidelines require that method names be verbs or verb phrases. Following this guideline also serves to distinguish method names from property and type names, which are noun or adjective phrases.

✓ **DO** give methods names that are verbs or verb phrases.

```
public class String {  
    public int CompareTo(...);  
    public string[] Split(...);  
    public string Trim();  
}
```

Names of Properties

Unlike other members, properties should be given noun phrase or adjective names. That is because a property refers to data, and the name of the property reflects that. PascalCasing is always used for property names.

✓ **DO** name properties using a noun, noun phrase, or adjective.

✗ **DO NOT** have properties that match the name of "Get" methods as in the following example:

```
public string TextWriter { get {...} set {...} }  
public string GetTextWriter(int value) { ... }
```

This pattern typically indicates that the property should really be a method.

✓ **DO** name collection properties with a plural phrase describing the items in the collection instead of using a singular phrase followed by "List" or "Collection."

✓ **DO** name Boolean properties with an affirmative phrase (**CanSeek** instead of **CantSeek**). Optionally, you can also prefix Boolean properties with "Is," "Can," or "Has," but only where it adds value.

✓ **CONSIDER** giving a property the same name as its type.

For example, the following property correctly gets and sets an enum value named **Color**, so the property is named **Color**:

```
public enum Color { ... }  
public class Control {  
    public Color Color { get {...} set {...} }  
}
```

Names of Events

Events always refer to some action, either one that is happening or one that has occurred. Therefore, as with methods, events are named with verbs, and verb tense is used to indicate the time when the event is raised.

✓ **DO** name events with a verb or a verb phrase.

Examples include **Clicked**, **Painting**, **DroppedDown**, and so on.

✓ **DO** give events names with a concept of before and after, using the present and past tenses.

For example, a close event that is raised before a window is closed would be called **Closing**, and one that is raised after the window is closed would be called **Closed**.

✗ **DO NOT** use "Before" or "After" prefixes or postfixes to indicate pre- and post-events. Use present and past tenses as just described.

✓ **DO** name event handlers (delegates used as types of events) with the "EventHandler" suffix, as shown in the following example:

```
public delegate void ClickedEventHandler(object sender, ClickedEventArgs e);
```

✓ **DO** use two parameters named *sender* and *e* in event handlers.

The sender parameter represents the object that raised the event. The sender parameter is typically of type **object**, even if it is possible to employ a more specific type.

✓ **DO** name event argument classes with the "EventArgs" suffix.

▲ Names of Fields

The field-naming guidelines apply to static public and protected fields. Internal and private fields are not covered by guidelines, and public or protected instance fields are not allowed by the [member design guidelines](#).

✓ **DO** use PascalCasing in field names.

✓ **DO** name fields using a noun, noun phrase, or adjective.

✗ **DO NOT** use a prefix for field names.

For example, do not use "g_" or "s_" to indicate static fields.

Portions © 2005, 2009 Microsoft Corporation. All rights reserved.

Reprinted by permission of Pearson Education, Inc. from [Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries, 2nd Edition](#) by Krzysztof Cwalina and Brad Abrams, published Oct 22, 2008 by Addison-Wesley Professional as part of the Microsoft Windows Development Series.

▲ See Also

Other Resources

[Framework Design Guidelines](#)

[Naming Guidelines](#)

Naming Parameters

.NET Framework 4.5

Beyond the obvious reason of readability, it is important to follow the guidelines for parameter names because parameters are displayed in documentation and in the designer when visual design tools provide Intellisense and class browsing functionality.

✓ **DO** use camelCasing in parameter names.

✓ **DO** use descriptive parameter names.

✓ **CONSIDER** using names based on a parameter's meaning rather than the parameter's type.

▲ Naming Operator Overload Parameters

✓ **DO** use *left* and *right* for binary operator overload parameter names if there is no meaning to the parameters.

✓ **DO** use *value* for unary operator overload parameter names if there is no meaning to the parameters.

✓ **CONSIDER** meaningful names for operator overload parameters if doing so adds significant value.

X **DO NOT** use abbreviations or numeric indices for operator overload parameter names.

Portions © 2005, 2009 Microsoft Corporation. All rights reserved.

Reprinted by permission of Pearson Education, Inc. from [Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries, 2nd Edition](#) by Krzysztof Cwalina and Brad Abrams, published Oct 22, 2008 by Addison-Wesley Professional as part of the Microsoft Windows Development Series.

▲ See Also

Other Resources

[Framework Design Guidelines](#)

[Naming Guidelines](#)

Naming Resources

.NET Framework 4.5

Because localizable resources can be referenced via certain objects as if they were properties, the naming guidelines for resources are similar to property guidelines.

✓ **DO** use PascalCasing in resource keys.

✓ **DO** provide descriptive rather than short identifiers.

✗ **DO NOT** use language-specific keywords of the main CLR languages.

✓ **DO** use only alphanumeric characters and underscores in naming resources.

✓ **DO** use the following naming convention for exception message resources.

The resource identifier should be the exception type name plus a short identifier of the exception:

`ArgumentExceptionIllegalCharacters`

`ArgumentExceptionInvalidName`

`ArgumentExceptionInvalidNameFormatted`

Portions © 2005, 2009 Microsoft Corporation. All rights reserved.

Reprinted by permission of Pearson Education, Inc. from *Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries, 2nd Edition* by Krzysztof Cwalina and Brad Abrams, published Oct 22, 2008 by Addison-Wesley Professional as part of the Microsoft Windows Development Series.

▲ See Also

Other Resources

[Framework Design Guidelines](#)

[Naming Guidelines](#)

Type Design Guidelines

.NET Framework 4.5

From the CLR perspective, there are only two categories of types—reference types and value types—but for the purpose of a discussion about framework design, we divide types into more logical groups, each with its own specific design rules.

Classes are the general case of reference types. They make up the bulk of types in the majority of frameworks. Classes owe their popularity to the rich set of object-oriented features they support and to their general applicability. Base classes and abstract classes are special logical groups related to extensibility.

Interfaces are types that can be implemented by both reference types and value types. They can thus serve as roots of polymorphic hierarchies of reference types and value types. In addition, interfaces can be used to simulate multiple inheritance, which is not natively supported by the CLR.

Structs are the general case of value types and should be reserved for small, simple types, similar to language primitives.

Enums are a special case of value types used to define short sets of values, such as days of the week, console colors, and so on.

Static classes are types intended to be containers for static members. They are commonly used to provide shortcuts to other operations.

Delegates, exceptions, attributes, arrays, and collections are all special cases of reference types intended for specific uses, and guidelines for their design and usage are discussed elsewhere in this book.

✓ **DO** ensure that each type is a well-defined set of related members, not just a random collection of unrelated functionality.

▲ In This Section

- [Choosing Between Class and Struct](#)
- [Abstract Class Design](#)
- [Static Class Design](#)
- [Interface Design](#)
- [Struct Design](#)
- [Enum Design](#)
- [Nested Types](#)

Portions © 2005, 2009 Microsoft Corporation. All rights reserved.

Reprinted by permission of Pearson Education, Inc. from [Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries, 2nd Edition](#) by Krzysztof Cwalina and Brad Abrams, published Oct 22, 2008 by Addison-Wesley Professional as part of the Microsoft Windows Development Series.

▲ See Also

- [Other Resources](#)
- [Framework Design Guidelines](#)

Choosing Between Class and Struct

.NET Framework 4.5

One of the basic design decisions every framework designer faces is whether to design a type as a class (a reference type) or as a struct (a value type). Good understanding of the differences in the behavior of reference types and value types is crucial in making this choice.

The first difference between reference types and value types we will consider is that reference types are allocated on the heap and garbage-collected, whereas value types are allocated either on the stack or inline in containing types and deallocated when the stack unwinds or when their containing type gets deallocated. Therefore, allocations and deallocations of value types are in general cheaper than allocations and deallocations of reference types.

Next, arrays of reference types are allocated out-of-line, meaning the array elements are just references to instances of the reference type residing on the heap. Value type arrays are allocated inline, meaning that the array elements are the actual instances of the value type. Therefore, allocations and deallocations of value type arrays are much cheaper than allocations and deallocations of reference type arrays. In addition, in a majority of cases value type arrays exhibit much better locality of reference.

The next difference is related to memory usage. Value types get boxed when cast to a reference type or one of the interfaces they implement. They get unboxed when cast back to the value type. Because boxes are objects that are allocated on the heap and are garbage-collected, too much boxing and unboxing can have a negative impact on the heap, the garbage collector, and ultimately the performance of the application. In contrast, no such boxing occurs as reference types are cast.

Next, reference type assignments copy the reference, whereas value type assignments copy the entire value. Therefore, assignments of large reference types are cheaper than assignments of large value types.

Finally, reference types are passed by reference, whereas value types are passed by value. Changes to an instance of a reference type affect all references pointing to the instance. Value type instances are copied when they are passed by value. When an instance of a value type is changed, it of course does not affect any of its copies. Because the copies are not created explicitly by the user but are implicitly created when arguments are passed or return values are returned, value types that can be changed can be confusing to many users. Therefore, value types should be immutable.

As a rule of thumb, the majority of types in a framework should be classes. There are, however, some situations in which the characteristics of a value type make it more appropriate to use structs.

✓ **CONSIDER** defining a struct instead of a class if instances of the type are small and commonly short-lived or are commonly embedded in other objects.

X **AVOID** defining a struct unless the type has all of the following characteristics:

- It logically represents a single value, similar to primitive types (**int**, **double**, etc.).
- It has an instance size under 16 bytes.
- It is immutable.
- It will not have to be boxed frequently.

In all other cases, you should define your types as classes.

Portions © 2005, 2009 Microsoft Corporation. All rights reserved.

Reprinted by permission of Pearson Education, Inc. from [Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries, 2nd Edition](#) by Krzysztof Cwalina and Brad Abrams, published Oct 22, 2008 by Addison-Wesley Professional as part of the Microsoft Windows Development Series.

▲ See Also

Other Resources

[Type Design Guidelines](#)

[Framework Design Guidelines](#)

Abstract Class Design

.NET Framework 4.5

X DO NOT define public or protected internal constructors in abstract types.

Constructors should be public only if users will need to create instances of the type. Because you cannot create instances of an abstract type, an abstract type with a public constructor is incorrectly designed and misleading to the users.

✓ DO define a protected or an internal constructor in abstract classes.

A protected constructor is more common and simply allows the base class to do its own initialization when subtypes are created.

An internal constructor can be used to limit concrete implementations of the abstract class to the assembly defining the class.

✓ DO provide at least one concrete type that inherits from each abstract class that you ship.

Doing this helps to validate the design of the abstract class. For example, [System.IO.FileStream](#) is an implementation of the [System.IO.Stream](#) abstract class.

Portions © 2005, 2009 Microsoft Corporation. All rights reserved.

Reprinted by permission of Pearson Education, Inc. from [Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries, 2nd Edition](#) by Krzysztof Cwalina and Brad Abrams, published Oct 22, 2008 by Addison-Wesley Professional as part of the Microsoft Windows Development Series.

▲ See Also

Other Resources

[Type Design Guidelines](#)

[Framework Design Guidelines](#)

Static Class Design

.NET Framework 4.5

A static class is defined as a class that contains only static members (of course besides the instance members inherited from [System.Object](#) and possibly a private constructor). Some languages provide built-in support for static classes. In C# 2.0 and later, when a class is declared to be static, it is sealed, abstract, and no instance members can be overridden or declared.

Static classes are a compromise between pure object-oriented design and simplicity. They are commonly used to provide shortcuts to other operations (such as [System.IO.File](#)), holders of extension methods, or functionality for which a full object-oriented wrapper is unwarranted (such as [System.Environment](#)).

✓ **DO** use static classes sparingly.

Static classes should be used only as supporting classes for the object-oriented core of the framework.

✗ **DO NOT** treat static classes as a miscellaneous bucket.

✗ **DO NOT** declare or override instance members in static classes.

✓ **DO** declare static classes as sealed, abstract, and add a private instance constructor if your programming language does not have built-in support for static classes.

Portions © 2005, 2009 Microsoft Corporation. All rights reserved.

Reprinted by permission of Pearson Education, Inc. from [Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries, 2nd Edition](#) by Krzysztof Cwalina and Brad Abrams, published Oct 22, 2008 by Addison-Wesley Professional as part of the Microsoft Windows Development Series.

▲ See Also

Other Resources

[Type Design Guidelines](#)

[Framework Design Guidelines](#)

Interface Design

.NET Framework 4.5

Although most APIs are best modeled using classes and structs, there are cases in which interfaces are more appropriate or are the only option.

The CLR does not support multiple inheritance (i.e., CLR classes cannot inherit from more than one base class), but it does allow types to implement one or more interfaces in addition to inheriting from a base class. Therefore, interfaces are often used to achieve the effect of multiple inheritance. For example, [IDisposable](#) is an interface that allows types to support disposability independent of any other inheritance hierarchy in which they want to participate.

The other situation in which defining an interface is appropriate is in creating a common interface that can be supported by several types, including some value types. Value types cannot inherit from types other than [ValueType](#), but they can implement interfaces, so using an interface is the only option in order to provide a common base type.

✓ **DO** define an interface if you need some common API to be supported by a set of types that includes value types.

✓ **CONSIDER** defining an interface if you need to support its functionality on types that already inherit from some other type.

✗ **AVOID** using marker interfaces (interfaces with no members).

If you need to mark a class as having a specific characteristic (marker), in general, use a custom attribute rather than an interface.

✓ **DO** provide at least one type that is an implementation of an interface.

Doing this helps to validate the design of the interface. For example, [List<T>](#) is an implementation of the [IList<T>](#) interface.

✓ **DO** provide at least one API that consumes each interface you define (a method taking the interface as a parameter or a property typed as the interface).

Doing this helps to validate the interface design. For example, [List<T>.Sort](#) consumes the [System.Collections.Generic.IComparer<T>](#) interface.

✗ **DO NOT** add members to an interface that has previously shipped.

Doing so would break implementations of the interface. You should create a new interface in order to avoid versioning problems.

Except for the situations described in these guidelines, you should, in general, choose classes rather than interfaces in designing managed code reusable libraries.

Portions © 2005, 2009 Microsoft Corporation. All rights reserved.

Reprinted by permission of Pearson Education, Inc. from [Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries, 2nd Edition](#) by Krzysztof Cwalina and Brad Abrams, published Oct 22, 2008 by Addison-Wesley Professional as part of the Microsoft Windows Development Series.

▲ See Also

Other Resources

[Type Design Guidelines](#)

[Framework Design Guidelines](#)

Struct Design

.NET Framework 4.5

The general-purpose value type is most often referred to as a struct, its C# keyword. This section provides guidelines for general struct design.

X DO NOT provide a default constructor for a struct.

Following this guideline allows arrays of structs to be created without having to run the constructor on each item of the array. Notice that C# does not allow structs to have default constructors.

X DO NOT define mutable value types.

Mutable value types have several problems. For example, when a property getter returns a value type, the caller receives a copy. Because the copy is created implicitly, developers might not be aware that they are mutating the copy, and not the original value. Also, some languages (dynamic languages, in particular) have problems using mutable value types because even local variables, when dereferenced, cause a copy to be made.

✓ **DO** ensure that a state where all instance data is set to zero, false, or null (as appropriate) is valid.

This prevents accidental creation of invalid instances when an array of the structs is created.

✓ **DO** implement [IEquatable<T>](#) on value types.

The [Object.Equals](#) method on value types causes boxing, and its default implementation is not very efficient, because it uses reflection. [Equals](#) can have much better performance and can be implemented so that it will not cause boxing.

X DO NOT explicitly extend [ValueType](#). In fact, most languages prevent this.

In general, structs can be very useful but should only be used for small, single, immutable values that will not be boxed frequently.

Portions © 2005, 2009 Microsoft Corporation. All rights reserved.

Reprinted by permission of Pearson Education, Inc. from [Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries, 2nd Edition](#) by Krzysztof Cwalina and Brad Abrams, published Oct 22, 2008 by Addison-Wesley Professional as part of the Microsoft Windows Development Series.

See Also

Concepts

[Choosing Between Class and Struct](#)

Other Resources

[Type Design Guidelines](#)

[Framework Design Guidelines](#)

Enum Design

.NET Framework 4.5

Enums are a special kind of value type. There are two kinds of enums: simple enums and flag enums.

Simple enums represent small closed sets of choices. A common example of the simple enum is a set of colors.

Flag enums are designed to support bitwise operations on the enum values. A common example of the flags enum is a list of options.

✓ **DO** use an enum to strongly type parameters, properties, and return values that represent sets of values.

✓ **DO** favor using an enum instead of static constants.

X DO NOT use an enum for open sets (such as the operating system version, names of your friends, etc.).

X DO NOT provide reserved enum values that are intended for future use.

You can always simply add values to the existing enum at a later stage. See [Adding Values to Enums](#) for more details on adding values to enums. Reserved values just pollute the set of real values and tend to lead to user errors.

X AVOID publicly exposing enums with only one value.

A common practice for ensuring future extensibility of C APIs is to add reserved parameters to method signatures. Such reserved parameters can be expressed as enums with a single default value. This should not be done in managed APIs. Method overloading allows adding parameters in future releases.

X DO NOT include sentinel values in enums.

Although they are sometimes helpful to framework developers, sentinel values are confusing to users of the framework. They are used to track the state of the enum rather than being one of the values from the set represented by the enum.

✓ **DO** provide a value of zero on simple enums.

Consider calling the value something like "None." If such a value is not appropriate for this particular enum, the most common default value for the enum should be assigned the underlying value of zero.

✓ **CONSIDER** using [Int32](#) (the default in most programming languages) as the underlying type of an enum unless any of the following is true:

- The enum is a flags enum and you have more than 32 flags, or expect to have more in the future.
- The underlying type needs to be different than [Int32](#) for easier interoperability with unmanaged code expecting different-size enums.
- A smaller underlying type would result in substantial savings in space. If you expect the enum to be used mainly as an argument for flow of control, the size makes little difference. The size savings might be significant if:
 - You expect the enum to be used as a field in a very frequently instantiated structure or class.
 - You expect users to create large arrays or collections of the enum instances.
 - You expect a large number of instances of the enum to be serialized.

For in-memory usage, be aware that managed objects are always **DWORD**-aligned, so you effectively need multiple enums or other small structures in an instance to pack a smaller enum with in order to make a difference, because the total instance size is always going to be rounded up to a **DWORD**.

✓ **DO** name flag enums with plural nouns or noun phrases and simple enums with singular nouns or noun phrases.

X DO NOT extend [System.Enum](#) directly.

[System.Enum](#) is a special type used by the CLR to create user-defined enumerations. Most programming languages provide a programming element that gives you access to this functionality. For example, in C# the **enum** keyword is used to define an enumeration.

▀ Designing Flag Enums

✓ **DO** apply the [System.FlagsAttribute](#) to flag enums. Do not apply this attribute to simple enums.

✓ **DO** use powers of two for the flag enum values so they can be freely combined using the bitwise OR operation.

✓ **CONSIDER** providing special enum values for commonly used combinations of flags.

Bitwise operations are an advanced concept and should not be required for simple tasks. [ReadWrite](#) is an example of such a special value.

X AVOID creating flag enums where certain combinations of values are invalid.

X AVOID using flag enum values of zero unless the value represents "all flags are cleared" and is named appropriately, as prescribed by the next guideline.

✓ **DO** name the zero value of flag enums **None**. For a flag enum, the value must always mean "all flags are cleared."

▀ Adding Value to Enums

It is very common to discover that you need to add values to an enum after you have already shipped it. There is a potential application compatibility problem when the

newly added value is returned from an existing API, because poorly written applications might not handle the new value correctly.

✓ **CONSIDER** adding values to enums, despite a small compatibility risk.

If you have real data about application incompatibilities caused by additions to an enum, consider adding a new API that returns the new and old values, and deprecate the old API, which should continue returning just the old values. This will ensure that your existing applications remain compatible.

Portions © 2005, 2009 Microsoft Corporation. All rights reserved.

Reprinted by permission of Pearson Education, Inc. from [Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries, 2nd Edition](#) by Krzysztof Cwalina and Brad Abrams, published Oct 22, 2008 by Addison-Wesley Professional as part of the Microsoft Windows Development Series.

▲ See Also

Other Resources

[Type Design Guidelines](#)

[Framework Design Guidelines](#)

Nested Types

.NET Framework 4.5

A nested type is a type defined within the scope of another type, which is called the enclosing type. A nested type has access to all members of its enclosing type. For example, it has access to private fields defined in the enclosing type and to protected fields defined in all ascendants of the enclosing type.

In general, nested types should be used sparingly. There are several reasons for this. Some developers are not fully familiar with the concept. These developers might, for example, have problems with the syntax of declaring variables of nested types. Nested types are also very tightly coupled with their enclosing types, and as such are not suited to be general-purpose types.

Nested types are best suited for modeling implementation details of their enclosing types. The end user should rarely have to declare variables of a nested type and almost never should have to explicitly instantiate nested types. For example, the enumerator of a collection can be a nested type of that collection. Enumerators are usually instantiated by their enclosing type, and because many languages support the `foreach` statement, enumerator variables rarely have to be declared by the end user.

✓ **DO** use nested types when the relationship between the nested type and its outer type is such that member-accessibility semantics are desirable.

✗ **DO NOT** use public nested types as a logical grouping construct; use namespaces for this.

✗ **AVOID** publicly exposed nested types. The only exception to this is if variables of the nested type need to be declared only in rare scenarios such as subclassing or other advanced customization scenarios.

✗ **DO NOT** use nested types if the type is likely to be referenced outside of the containing type.

For example, an enum passed to a method defined on a class should not be defined as a nested type in the class.

✗ **DO NOT** use nested types if they need to be instantiated by client code. If a type has a public constructor, it should probably not be nested.

If a type can be instantiated, that seems to indicate the type has a place in the framework on its own (you can create it, work with it, and destroy it without ever using the outer type), and thus should not be nested. Inner types should not be widely reused outside of the outer type without any relationship whatsoever to the outer type.

✗ **DO NOT** define a nested type as a member of an interface. Many languages do not support such a construct.

Portions © 2005, 2009 Microsoft Corporation. All rights reserved.

Reprinted by permission of Pearson Education, Inc. from [Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries, 2nd Edition](#) by Krzysztof Cwalina and Brad Abrams, published Oct 22, 2008 by Addison-Wesley Professional as part of the Microsoft Windows Development Series.

▲ See Also

Other Resources

[Type Design Guidelines](#)

[Framework Design Guidelines](#)

Member Design Guidelines

.NET Framework 4.5

Methods, properties, events, constructors, and fields are collectively referred to as members. Members are ultimately the means by which framework functionality is exposed to the end users of a framework.

Members can be virtual or nonvirtual, concrete or abstract, static or instance, and can have several different scopes of accessibility. All this variety provides incredible expressiveness but at the same time requires care on the part of the framework designer.

This chapter offers basic guidelines that should be followed when designing members of any type.

▲ In This Section

- [Member Overloading](#)
- [Property Design](#)
- [Constructor Design](#)
- [Event Design](#)
- [Field Design](#)
- [Extension Methods](#)
- [Operator Overloads](#)
- [Parameter Design](#)

Portions © 2005, 2009 Microsoft Corporation. All rights reserved.

Reprinted by permission of Pearson Education, Inc. from [Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries, 2nd Edition](#) by Krzysztof Cwalina and Brad Abrams, published Oct 22, 2008 by Addison-Wesley Professional as part of the Microsoft Windows Development Series.

▲ See Also

- [Other Resources](#)
- [Framework Design Guidelines](#)

Member Overloading

.NET Framework 4.5

Member overloading means creating two or more members on the same type that differ only in the number or type of parameters but have the same name. For example, in the following, the `WriteLine` method is overloaded:

```
public static class Console {  
    public void WriteLine();  
    public void WriteLine(string value);  
    public void WriteLine(bool value);  
    ...  
}
```

Because only methods, constructors, and indexed properties can have parameters, only those members can be overloaded.

Overloading is one of the most important techniques for improving usability, productivity, and readability of reusable libraries. Overloading on the number of parameters makes it possible to provide simpler versions of constructors and methods. Overloading on the parameter type makes it possible to use the same member name for members performing identical operations on a selected set of different types.

✓ **DO** try to use descriptive parameter names to indicate the default used by shorter overloads.

✗ **AVOID** arbitrarily varying parameter names in overloads. If a parameter in one overload represents the same input as a parameter in another overload, the parameters should have the same name.

✗ **AVOID** being inconsistent in the ordering of parameters in overloaded members. Parameters with the same name should appear in the same position in all overloads.

✓ **DO** make only the longest overload virtual (if extensibility is required). Shorter overloads should simply call through to a longer overload.

✗ **DO NOT** use `ref` or `out` modifiers to overload members.

Some languages cannot resolve calls to overloads like this. In addition, such overloads usually have completely different semantics and probably should not be overloads but two separate methods instead.

✗ **DO NOT** have overloads with parameters at the same position and similar types yet with different semantics.

✓ **DO** allow `null` to be passed for optional arguments.

✓ **DO** use member overloading rather than defining members with default arguments.

Default arguments are not CLS compliant.

Portions © 2005, 2009 Microsoft Corporation. All rights reserved.

Reprinted by permission of Pearson Education, Inc. from *Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries, 2nd Edition* by Krzysztof Cwalina and Brad Abrams, published Oct 22, 2008 by Addison-Wesley Professional as part of the Microsoft Windows Development Series.

▲ See Also

Other Resources

[Member Design Guidelines](#)

[Framework Design Guidelines](#)

Property Design

.NET Framework 4.5

Although properties are technically very similar to methods, they are quite different in terms of their usage scenarios. They should be seen as smart fields. They have the calling syntax of fields, and the flexibility of methods.

✓ **DO** create get-only properties if the caller should not be able to change the value of the property.

Keep in mind that if the type of the property is a mutable reference type, the property value can be changed even if the property is get-only.

✗ **DO NOT** provide set-only properties or properties with the setter having broader accessibility than the getter.

For example, do not use properties with a public setter and a protected getter.

If the property getter cannot be provided, implement the functionality as a method instead. Consider starting the method name with **Set** and follow with what you would have named the property. For example, `AppDomain` has a method called `SetCachePath` instead of having a set-only property called `CachePath`.

✓ **DO** provide sensible default values for all properties, ensuring that the defaults do not result in a security hole or terribly inefficient code.

✓ **DO** allow properties to be set in any order even if this results in a temporary invalid state of the object.

It is common for two or more properties to be interrelated to a point where some values of one property might be invalid given the values of other properties on the same object. In such cases, exceptions resulting from the invalid state should be postponed until the interrelated properties are actually used together by the object.

✓ **DO** preserve the previous value if a property setter throws an exception.

✗ **AVOID** throwing exceptions from property getters.

Property getters should be simple operations and should not have any preconditions. If a getter can throw an exception, it should probably be redesigned to be a method. Notice that this rule does not apply to indexers, where we do expect exceptions as a result of validating the arguments.

Indexed Property Design

An indexed property is a special property that can have parameters and can be called with special syntax similar to array indexing.

Indexed properties are commonly referred to as indexers. Indexers should be used only in APIs that provide access to items in a logical collection. For example, a string is a collection of characters, and the indexer on `System.String` was added to access its characters.

✓ **CONSIDER** using indexers to provide access to data stored in an internal array.

✓ **CONSIDER** providing indexers on types representing collections of items.

✗ **AVOID** using indexed properties with more than one parameter.

If the design requires multiple parameters, reconsider whether the property really represents an accessor to a logical collection. If it does not, use methods instead. Consider starting the method name with **Get** or **Set**.

✗ **AVOID** indexers with parameter types other than `System.Int32`, `System.Int64`, `System.String`, `System.Object`, or an enum.

If the design requires other types of parameters, strongly reevaluate whether the API really represents an accessor to a logical collection. If it does not, use a method. Consider starting the method name with **Get** or **Set**.

✓ **DO** use the name `Item` for indexed properties unless there is an obviously better name (e.g., see the `Chars` property on `System.String`).

In C#, indexers are by default named `Item`. The `IndexerNameAttribute` can be used to customize this name.

✗ **DO NOT** provide both an indexer and methods that are semantically equivalent.

✗ **DO NOT** provide more than one family of overloaded indexers in one type.

This is enforced by the C# compiler.

✗ **DO NOT** use nondefault indexed properties.

This is enforced by the C# compiler.

Property Change Notification Events

Sometimes it is useful to provide an event notifying the user of changes in a property value. For example, `System.Windows.Forms.Control` raises a `TextChanged` event after the value of its `Text` property has changed.

✓ **CONSIDER** raising change notification events when property values in high-level APIs (usually designer components) are modified.

If there is a good scenario for a user to know when a property of an object is changing, the object should raise a change notification event for the property.

However, it is unlikely to be worth the overhead to raise such events for low-level APIs such as base types or collections. For example, `List<T>` would not raise such events when a new item is added to the list and the `Count` property changes.

✓ **CONSIDER** raising change notification events when the value of a property changes via external forces.

If a property value changes via some external force (in a way other than by calling methods on the object), raise events indicate to the developer that the value is changing and has changed. A good example is the `Text` property of a text box control. When the user types text in a `TextBox`, the property value automatically changes.

Portions © 2005, 2009 Microsoft Corporation. All rights reserved.

Reprinted by permission of Pearson Education, Inc. from [Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries, 2nd Edition](#) by Krzysztof Cwalina and Brad Abrams, published Oct 22, 2008 by Addison-Wesley Professional as part of the Microsoft Windows Development Series.

▲ See Also

Other Resources

[Member Design Guidelines](#)

[Framework Design Guidelines](#)

Constructor Design

.NET Framework 4.5

There are two kinds of constructors: type constructors and instance constructors.

Type constructors are static and are run by the CLR before the type is used. Instance constructors run when an instance of a type is created.

Type constructors cannot take any parameters. Instance constructors can. Instance constructors that don't take any parameters are often called default constructors.

Constructors are the most natural way to create instances of a type. Most developers will search and try to use a constructor before they consider alternative ways of creating instances (such as factory methods).

✓ **CONSIDER** providing simple, ideally default, constructors.

A simple constructor has a very small number of parameters, and all parameters are primitives or enums. Such simple constructors increase usability of the framework.

✓ **CONSIDER** using a static factory method instead of a constructor if the semantics of the desired operation do not map directly to the construction of a new instance, or if following the constructor design guidelines feels unnatural.

✓ **DO** use constructor parameters as shortcuts for setting main properties.

There should be no difference in semantics between using the empty constructor followed by some property sets and using a constructor with multiple arguments.

✓ **DO** use the same name for constructor parameters and a property if the constructor parameters are used to simply set the property.

The only difference between such parameters and the properties should be casing.

✓ **DO** minimal work in the constructor.

Constructors should not do much work other than capture the constructor parameters. The cost of any other processing should be delayed until required.

✓ **DO** throw exceptions from instance constructors, if appropriate.

✓ **DO** explicitly declare the public default constructor in classes, if such a constructor is required.

If you don't explicitly declare any constructors on a type, many languages (such as C#) will automatically add a public default constructor. (Abstract classes get a protected constructor.)

Adding a parameterized constructor to a class prevents the compiler from adding the default constructor. This often causes accidental breaking changes.

✗ **AVOID** explicitly defining default constructors on structs.

This makes array creation faster, because if the default constructor is not defined, it does not have to be run on every slot in the array. Note that many compilers, including C#, don't allow structs to have parameterless constructors for this reason.

✗ **AVOID** calling virtual members on an object inside its constructor.

Calling a virtual member will cause the most derived override to be called, even if the constructor of the most derived type has not been fully run yet.

▀ Type Constructor Guidelines

✓ **DO** make static constructors private.

A static constructor, also called a class constructor, is used to initialize a type. The CLR calls the static constructor before the first instance of the type is created or any static members on that type are called. The user has no control over when the static constructor is called. If a static constructor is not private, it can be called by code other than the CLR. Depending on the operations performed in the constructor, this can cause unexpected behavior. The C# compiler forces static constructors to be private.

✗ **DO NOT** throw exceptions from static constructors.

If an exception is thrown from a type constructor, the type is not usable in the current application domain.

✓ **CONSIDER** initializing static fields inline rather than explicitly using static constructors, because the runtime is able to optimize the performance of types that don't have an explicitly defined static constructor.

Portions © 2005, 2009 Microsoft Corporation. All rights reserved.

Reprinted by permission of Pearson Education, Inc. from [Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries, 2nd Edition](#) by Krzysztof Cwalina and Brad Abrams, published Oct 22, 2008 by Addison-Wesley Professional as part of the Microsoft Windows Development Series.

▀ See Also

Other Resources

[Member Design Guidelines](#)

[Framework Design Guidelines](#)

Event Design

.NET Framework 4.5

Events are the most commonly used form of callbacks (constructs that allow the framework to call into user code). Other callback mechanisms include members taking delegates, virtual members, and interface-based plug-ins. Data from usability studies indicate that the majority of developers are more comfortable using events than they are using the other callback mechanisms. Events are nicely integrated with Visual Studio and many languages.

It is important to note that there are two groups of events: events raised before a state of the system changes, called pre-events, and events raised after a state changes, called post-events. An example of a pre-event would be `Form.Closing`, which is raised before a form is closed. An example of a post-event would be `Form.Closed`, which is raised after a form is closed.

✓ **DO** use the term "raise" for events rather than "fire" or "trigger."

✓ **DO** use `System.EventHandler<EventArgs>` instead of manually creating new delegates to be used as event handlers.

✓ **CONSIDER** using a subclass of `EventArgs` as the event argument, unless you are absolutely sure the event will never need to carry any data to the event handling method, in which case you can use the `EventArgs` type directly.

If you ship an API using `EventArgs` directly, you will never be able to add any data to be carried with the event without breaking compatibility. If you use a subclass, even if initially completely empty, you will be able to add properties to the subclass when needed.

✓ **DO** use a protected virtual method to raise each event. This is only applicable to nonstatic events on unsealed classes, not to structs, sealed classes, or static events.

The purpose of the method is to provide a way for a derived class to handle the event using an override. Overriding is a more flexible, faster, and more natural way to handle base class events in derived classes. By convention, the name of the method should start with "On" and be followed with the name of the event.

The derived class can choose not to call the base implementation of the method in its override. Be prepared for this by not including any processing in the method that is required for the base class to work correctly.

✓ **DO** take one parameter to the protected method that raises an event.

The parameter should be named `e` and should be typed as the event argument class.

✗ **DO NOT** pass null as the sender when raising a nonstatic event.

✓ **DO** pass null as the sender when raising a static event.

✗ **DO NOT** pass null as the event data parameter when raising an event.

You should pass `EventArgs.Empty` if you don't want to pass any data to the event handling method. Developers expect this parameter not to be null.

✓ **CONSIDER** raising events that the end user can cancel. This only applies to pre-events.

Use `System.ComponentModel.CancelEventArgs` or its subclass as the event argument to allow the end user to cancel events.

Custom Event Handler Design

There are cases in which `EventHandler<T>` cannot be used, such as when the framework needs to work with earlier versions of the CLR, which did not support Generics. In such cases, you might need to design and develop a custom event handler delegate.

✓ **DO** use a return type of void for event handlers.

An event handler can invoke multiple event handling methods, possibly on multiple objects. If event handling methods were allowed to return a value, there would be multiple return values for each event invocation.

✓ **DO** use `object` as the type of the first parameter of the event handler, and call it `sender`.

✓ **DO** use `System.EventArgs` or its subclass as the type of the second parameter of the event handler, and call it `e`.

✗ **DO NOT** have more than two parameters on event handlers.

Portions © 2005, 2009 Microsoft Corporation. All rights reserved.

Reprinted by permission of Pearson Education, Inc. from [Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries, 2nd Edition](#) by Krzysztof Cwalina and Brad Abrams, published Oct 22, 2008 by Addison-Wesley Professional as part of the Microsoft Windows Development Series.

See Also

Other Resources

[Member Design Guidelines](#)

[Framework Design Guidelines](#)

Field Design

.NET Framework 4.5

The principle of encapsulation is one of the most important notions in object-oriented design. This principle states that data stored inside an object should be accessible only to that object.

A useful way to interpret the principle is to say that a type should be designed so that changes to fields of that type (name or type changes) can be made without breaking code other than for members of the type. This interpretation immediately implies that all fields must be private.

We exclude constant and static read-only fields from this strict restriction, because such fields, almost by definition, are never required to change.

X DO NOT provide instance fields that are public or protected.

You should provide properties for accessing fields instead of making them public or protected.

✓ **DO** use constant fields for constants that will never change.

The compiler burns the values of const fields directly into calling code. Therefore, const values can never be changed without the risk of breaking compatibility.

✓ **DO** use public static **readonly** fields for predefined object instances.

If there are predefined instances of the type, declare them as public read-only static fields of the type itself.

X DO NOT assign instances of mutable types to **readonly** fields.

A mutable type is a type with instances that can be modified after they are instantiated. For example, arrays, most collections, and streams are mutable types, but [System.Int32](#), [System.Uri](#), and [System.String](#) are all immutable. The read-only modifier on a reference type field prevents the instance stored in the field from being replaced, but it does not prevent the field's instance data from being modified by calling members changing the instance.

Portions © 2005, 2009 Microsoft Corporation. All rights reserved.

Reprinted by permission of Pearson Education, Inc. from [Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries, 2nd Edition](#) by Krzysztof Cwalina and Brad Abrams, published Oct 22, 2008 by Addison-Wesley Professional as part of the Microsoft Windows Development Series.

▲ See Also

Other Resources

[Member Design Guidelines](#)

[Framework Design Guidelines](#)

Extension Methods

.NET Framework 4.5

Extension methods are a language feature that allows static methods to be called using instance method call syntax. These methods must take at least one parameter, which represents the instance the method is to operate on.

The class that defines such extension methods is referred to as the "sponsor" class, and it must be declared as static. To use extension methods, one must import the namespace defining the sponsor class.

X AVOID frivolously defining extension methods, especially on types you don't own.

If you do own source code of a type, consider using regular instance methods instead. If you don't own, and you want to add a method, be very careful. Liberal use of extension methods has the potential of cluttering APIs of types that were not designed to have these methods.

✓ **CONSIDER** using extension methods in any of the following scenarios:

- To provide helper functionality relevant to every implementation of an interface, if said functionality can be written in terms of the core interface. This is because concrete implementations cannot otherwise be assigned to interfaces. For example, the `LINQ to Objects` operators are implemented as extension methods for all `IEnumerable<T>` types. Thus, any `IEnumerable<T>` implementation is automatically LINQ-enabled.
- When an instance method would introduce a dependency on some type, but such a dependency would break dependency management rules. For example, a dependency from `String` to `System.Uri` is probably not desirable, and so `String.ToUri()` instance method returning `System.Uri` would be the wrong design from a dependency management perspective. A static extension method `Uri.ToUri(this string str)` returning `System.Uri` would be a much better design.

X AVOID defining extension methods on `System.Object`.

VB users will not be able to call such methods on object references using the extension method syntax. VB does not support calling such methods because, in VB, declaring a reference as `Object` forces all method invocations on it to be late bound (actual member called is determined at runtime), while bindings to extension methods are determined at compile-time (early bound).

Note that the guideline applies to other languages where the same binding behavior is present, or where extension methods are not supported.

X DO NOT put extension methods in the same namespace as the extended type unless it is for adding methods to interfaces or for dependency management.

X AVOID defining two or more extension methods with the same signature, even if they reside in different namespaces.

✓ **CONSIDER** defining extension methods in the same namespace as the extended type if the type is an interface and if the extension methods are meant to be used in most or all cases.

X DO NOT define extension methods implementing a feature in namespaces normally associated with other features. Instead, define them in the namespace associated with the feature they belong to.

X AVOID generic naming of namespaces dedicated to extension methods (e.g., "Extensions"). Use a descriptive name (e.g., "Routing") instead.

Portions © 2005, 2009 Microsoft Corporation. All rights reserved.

Reprinted by permission of Pearson Education, Inc. from [Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries, 2nd Edition](#) by Krzysztof Cwalina and Brad Abrams, published Oct 22, 2008 by Addison-Wesley Professional as part of the Microsoft Windows Development Series.

▲ See Also

Other Resources

[Member Design Guidelines](#)

[Framework Design Guidelines](#)

Operator Overloads

.NET Framework 4.5

Operator overloads allow framework types to appear as if they were built-in language primitives.

Although allowed and useful in some situations, operator overloads should be used cautiously. There are many cases in which operator overloading has been abused, such as when framework designers started to use operators for operations that should be simple methods. The following guidelines should help you decide when and how to use operator overloading.

X AVOID defining operator overloads, except in types that should feel like primitive (built-in) types.

✓ CONSIDER defining operator overloads in a type that should feel like a primitive type.

For example, `System.String` has `operator==` and `operator!=` defined.

✓ DO define operator overloads in structs that represent numbers (such as `System.Decimal`).

X DO NOT be cute when defining operator overloads.

Operator overloading is useful in cases in which it is immediately obvious what the result of the operation will be. For example, it makes sense to be able to subtract one `DateTime` from another `DateTime` and get a `TimeSpan`. However, it is not appropriate to use the logical union operator to union two database queries, or to use the shift operator to write to a stream.

X DO NOT provide operator overloads unless at least one of the operands is of the type defining the overload.

✓ DO overload operators in a symmetric fashion.

For example, if you overload the `operator==`, you should also overload the `operator!=`. Similarly, if you overload the `operator<`, you should also overload the `operator>`, and so on.

✓ CONSIDER providing methods with friendly names that correspond to each overloaded operator.

Many languages do not support operator overloading. For this reason, it is recommended that types that overload operators include a secondary method with an appropriate domain-specific name that provides equivalent functionality.

The following table contains a list of operators and the corresponding friendly method names.

C# Operator Symbol	Metadata Name	Friendly Name
N/A	op_Implicit	To<TypeName>/From<TypeName>
N/A	op_Explicit	To<TypeName>/From<TypeName>
+ (binary)	op_Addition	Add
- (binary)	op_Subtraction	Subtract
* (binary)	op_Multiply	Multiply
/	op_Division	Divide
%	op_Modulus	Mod or Remainder
^	op_ExclusiveOr	Xor
& (binary)	op_BitwiseAnd	BitwiseAnd
	op_BitwiseOr	BitwiseOr
&&	op_LogicalAnd	And
	op_LogicalOr	Or
=	op_Assign	Assign
<<	op_LeftShift	LeftShift
>>	op_RightShift	RightShift
N/A	op_SignedRightShift	SignedRightShift
N/A	op_UnsignedRightShift	UnsignedRightShift
==	op_Equality	Equals
!=	op_Inequality	Equals

>	op_GreaterThan	CompareTo
<	op_LessThan	CompareTo
>=	op_GreaterThanOrEqual	CompareTo
<=	op_LessThanOrEqual	CompareTo
*=	op_MultiplicationAssignment	Multiply
-=	op_SubtractionAssignment	Subtract
^=	op_ExclusiveOrAssignment	Xor
<<=	op_LeftShiftAssignment	LeftShift
%=	op_ModulusAssignment	Mod
+=	op_AdditionAssignment	Add
&=	op_BitwiseAndAssignment	BitwiseAnd
=	op_BitwiseOrAssignment	BitwiseOr
,	op_Comma	Comma
/=	op_DivisionAssignment	Divide
--	op_Decrement	Decrement
++	op_Increment	Increment
- (unary)	op_UnaryNegation	Negate
+ (unary)	op_UnaryPlus	Plus
~	op_OnesComplement	OnesComplement

▲ Overloading Operator ==

Overloading `operator ==` is quite complicated. The semantics of the operator need to be compatible with several other members, such as [Object.Equals](#).

▲ Conversion Operators

Conversion operators are unary operators that allow conversion from one type to another. The operators must be defined as static members on either the operand or the return type. There are two types of conversion operators: implicit and explicit.

X DO NOT provide a conversion operator if such conversion is not clearly expected by the end users.

X DO NOT define conversion operators outside of a type's domain.

For example, [Int32](#), [Double](#), and [Decimal](#) are all numeric types, whereas [DateTime](#) is not. Therefore, there should be no conversion operator to convert a [Double\(long\)](#) to a [DateTime](#). A constructor is preferred in such a case.

X DO NOT provide an implicit conversion operator if the conversion is potentially lossy.

For example, there should not be an implicit conversion from [Double](#) to [Int32](#) because [Double](#) has a wider range than [Int32](#). An explicit conversion operator can be provided even if the conversion is potentially lossy.

X DO NOT throw exceptions from implicit casts.

It is very difficult for end users to understand what is happening, because they might not be aware that a conversion is taking place.

✓ **DO** throw [System.InvalidCastException](#) if a call to a cast operator results in a lossy conversion and the contract of the operator does not allow lossy conversions.

Portions © 2005, 2009 Microsoft Corporation. All rights reserved.

Reprinted by permission of Pearson Education, Inc. from [Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries, 2nd Edition](#) by Krzysztof Cwalina and Brad Abrams, published Oct 22, 2008 by Addison-Wesley Professional as part of the Microsoft Windows Development Series.

▲ See Also

Other Resources
[Member Design Guidelines](#)

Parameter Design

.NET Framework 4.5

This section provides broad guidelines on parameter design, including sections with guidelines for checking arguments. In addition, you should refer to the guidelines described in [Naming Parameters](#).

✓ **DO** use the least derived parameter type that provides the functionality required by the member.

For example, suppose you want to design a method that enumerates a collection and prints each item to the console. Such a method should take [IEnumerable](#) as the parameter, not [ArrayList](#) or [IList](#), for example.

✗ **DO NOT** use reserved parameters.

If more input to a member is needed in some future version, a new overload can be added.

✗ **DO NOT** have publicly exposed methods that take pointers, arrays of pointers, or multidimensional arrays as parameters.

Pointers and multidimensional arrays are relatively difficult to use properly. In almost all cases, APIs can be redesigned to avoid taking these types as parameters.

✓ **DO** place all [out](#) parameters following all of the by-value and [ref](#) parameters (excluding parameter arrays), even if it results in an inconsistency in parameter ordering between overloads (see [Member Overloading](#)).

The [out](#) parameters can be seen as extra return values, and grouping them together makes the method signature easier to understand.

✓ **DO** be consistent in naming parameters when overriding members or implementing interface members.

This better communicates the relationship between the methods.

▀ Choosing Between Enum and Boolean Parameters

✓ **DO** use enums if a member would otherwise have two or more Boolean parameters.

✗ **DO NOT** use Booleans unless you are absolutely sure there will never be a need for more than two values.

Enums give you some room for future addition of values, but you should be aware of all the implications of adding values to enums, which are described in [Enum Design](#).

✓ **CONSIDER** using Booleans for constructor parameters that are truly two-state values and are simply used to initialize Boolean properties.

▀ Validating Arguments

✓ **DO** validate arguments passed to public, protected, or explicitly implemented members. Throw [System.ArgumentException](#), or one of its subclasses, if the validation fails.

Note that the actual validation does not necessarily have to happen in the public or protected member itself. It could happen at a lower level in some private or internal routine. The main point is that the entire surface area that is exposed to the end users checks the arguments.

✓ **DO** throw [ArgumentNullException](#) if a null argument is passed and the member does not support null arguments.

✓ **DO** validate enum parameters.

Do not assume enum arguments will be in the range defined by the enum. The CLR allows casting any integer value into an enum value even if the value is not defined in the enum.

✗ **DO NOT** use [Enum.IsDefined](#) for enum range checks.

✓ **DO** be aware that mutable arguments might have changed after they were validated.

If the member is security sensitive, you are encouraged to make a copy and then validate and process the argument.

▀ Parameter Passing

From the perspective of a framework designer, there are three main groups of parameters: by-value parameters, [ref](#) parameters, and [out](#) parameters.

When an argument is passed through a by-value parameter, the member receives a copy of the actual argument passed in. If the argument is a value type, a copy of the argument is put on the stack. If the argument is a reference type, a copy of the reference is put on the stack. Most popular CLR languages, such as C#, VB.NET, and C++, default to passing parameters by value.

When an argument is passed through a [ref](#) parameter, the member receives a reference to the actual argument passed in. If the argument is a value type, a reference to the argument is put on the stack. If the argument is a reference type, a reference to the reference is put on the stack. [Ref](#) parameters can be used to allow the member to modify arguments passed by the caller.

[Out](#) parameters are similar to [ref](#) parameters, with some small differences. The parameter is initially considered unassigned and cannot be read in the member body before it is assigned some value. Also, the parameter has to be assigned some value before the member returns.

✗ **AVOID** using [out](#) or [ref](#) parameters.

Using **out** or **ref** parameters requires experience with pointers, understanding how value types and reference types differ, and handling methods with multiple return values. Also, the difference between **out** and **ref** parameters is not widely understood. Framework architects designing for a general audience should not expect users to master working with **out** or **ref** parameters.

X DO NOT pass reference types by reference.

There are some limited exceptions to the rule, such as a method that can be used to swap references.

Members with Variable Number of Parameters

Members that can take a variable number of arguments are expressed by providing an array parameter. For example, [String](#) provides the following method:

```
public class String {  
    public static string Format(string format, object[] parameters);  
}
```

A user can then call the [String.Format](#) method, as follows:

```
String.Format("File {0} not found in {1}", new object[] {filename, directory});
```

Adding the C# **params** keyword to an array parameter changes the parameter to a so-called params array parameter and provides a shortcut to creating a temporary array.

```
public class String {  
    public static string Format(string format, params object[] parameters);  
}
```

Doing this allows the user to call the method by passing the array elements directly in the argument list.

```
String.Format("File {0} not found in {1}", filename, directory);
```

Note that the **params** keyword can be added only to the last parameter in the parameter list.

✓ **CONSIDER** adding the **params** keyword to array parameters if you expect the end users to pass arrays with a small number of elements. If it's expected that lots of elements will be passed in common scenarios, users will probably not pass these elements inline anyway, and so the **params** keyword is not necessary.

X AVOID using params arrays if the caller would almost always have the input already in an array.

For example, members with byte array parameters would almost never be called by passing individual bytes. For this reason, byte array parameters in the .NET Framework do not use the **params** keyword.

X DO NOT use params arrays if the array is modified by the member taking the params array parameter.

Because of the fact that many compilers turn the arguments to the member into a temporary array at the call site, the array might be a temporary object, and therefore any modifications to the array will be lost.

✓ **CONSIDER** using the **params** keyword in a simple overload, even if a more complex overload could not use it.

Ask yourself if users would value having the params array in one overload even if it wasn't in all overloads.

✓ **DO** try to order parameters to make it possible to use the **params** keyword.

✓ **CONSIDER** providing special overloads and code paths for calls with a small number of arguments in extremely performance-sensitive APIs.

This makes it possible to avoid creating array objects when the API is called with a small number of arguments. Form the names of the parameters by taking a singular form of the array parameter and adding a numeric suffix.

You should only do this if you are going to special-case the entire code path, not just create an array and call the more general method.

✓ **DO** be aware that null could be passed as a params array argument.

You should validate that the array is not null before processing.

X DO NOT use the **varargs** methods, otherwise known as the ellipsis.

Some CLR languages, such as C++, support an alternative convention for passing variable parameter lists called **varargs** methods. The convention should not be used in frameworks, because it is not CLS compliant.

Pointer Parameters

In general, pointers should not appear in the public surface area of a well-designed managed code framework. Most of the time, pointers should be encapsulated. However, in some cases pointers are required for interoperability reasons, and using pointers in such cases is appropriate.

✓ **DO** provide an alternative for any member that takes a pointer argument, because pointers are not CLS-compliant.

X AVOID doing expensive argument checking of pointer arguments.

✓ **DO** follow common pointer-related conventions when designing members with pointers.

For example, there is no need to pass the start index, because simple pointer arithmetic can be used to accomplish the same result.

Portions © 2005, 2009 Microsoft Corporation. All rights reserved.

Reprinted by permission of Pearson Education, Inc. from [Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries, 2nd Edition](#) by Krzysztof Cwalina and Brad Abrams, published Oct 22, 2008 by Addison-Wesley Professional as part of the Microsoft Windows Development Series.

▲ See Also

Other Resources

[Member Design Guidelines](#)

[Framework Design Guidelines](#)

Designing for Extensibility

.NET Framework 4.5

One important aspect of designing a framework is making sure the extensibility of the framework has been carefully considered. This requires that you understand the costs and benefits associated with various extensibility mechanisms. This chapter helps you decide which of the extensibility mechanisms—subclassing, events, virtual members, callbacks, and so on—can best meet the requirements of your framework.

There are many ways to allow extensibility in frameworks. They range from less powerful but less costly to very powerful but expensive. For any given extensibility requirement, you should choose the least costly extensibility mechanism that meets the requirements. Keep in mind that it's usually possible to add more extensibility later, but you can never take it away without introducing breaking changes.

▲ In This Section

- [Unsealed Classes](#)
- [Protected Members](#)
- [Events and Callbacks](#)
- [Virtual Members](#)
- [Abstractions \(Abstract Types and Interfaces\)](#)
- [Base Classes for Implementing Abstractions](#)
- [Sealing](#)

Portions © 2005, 2009 Microsoft Corporation. All rights reserved.

Reprinted by permission of Pearson Education, Inc. from [Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries, 2nd Edition](#) by Krzysztof Cwalina and Brad Abrams, published Oct 22, 2008 by Addison-Wesley Professional as part of the Microsoft Windows Development Series.

▲ See Also

- [Other Resources](#)
- [Framework Design Guidelines](#)

Unsealed Classes

.NET Framework 4.5

Sealed classes cannot be inherited from, and they prevent extensibility. In contrast, classes that can be inherited from are called unsealed classes.

✓ **CONSIDER** using unsealed classes with no added virtual or protected members as a great way to provide inexpensive yet much appreciated extensibility to a framework.

Developers often want to inherit from unsealed classes so as to add convenience members such as custom constructors, new methods, or method overloads. For example, `System.Messaging.MessageQueue` is unsealed and thus allows users to create custom queues that default to a particular queue path or to add custom methods that simplify the API for specific scenarios.

Classes are unsealed by default in most programming languages, and this is also the recommended default for most classes in frameworks. The extensibility afforded by unsealed types is much appreciated by framework users and quite inexpensive to provide because of relatively low test costs associated with unsealed types.

Portions © 2005, 2009 Microsoft Corporation. All rights reserved.

Reprinted by permission of Pearson Education, Inc. from [Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries, 2nd Edition](#) by Krzysztof Cwalina and Brad Abrams, published Oct 22, 2008 by Addison-Wesley Professional as part of the Microsoft Windows Development Series.

▲ See Also

Concepts

[Sealing](#)

Other Resources

[Framework Design Guidelines](#)

[Designing for Extensibility](#)

Protected Members

.NET Framework 4.5

Protected members by themselves do not provide any extensibility, but they can make extensibility through subclassing more powerful. They can be used to expose advanced customization options without unnecessarily complicating the main public interface.

Framework designers need to be careful with protected members because the name "protected" can give a false sense of security. Anyone is able to subclass an unsealed class and access protected members, and so all the same defensive coding practices used for public members apply to protected members.

✓ **CONSIDER** using protected members for advanced customization.

✓ **DO** treat protected members in unsealed classes as public for the purpose of security, documentation, and compatibility analysis.

Anyone can inherit from a class and access the protected members.

Portions © 2005, 2009 Microsoft Corporation. All rights reserved.

Reprinted by permission of Pearson Education, Inc. from [Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries, 2nd Edition](#) by Krzysztof Cwalina and Brad Abrams, published Oct 22, 2008 by Addison-Wesley Professional as part of the Microsoft Windows Development Series.

▲ See Also

Other Resources

[Framework Design Guidelines](#)

[Designing for Extensibility](#)

Events and Callbacks

.NET Framework 4.5

Callbacks are extensibility points that allow a framework to call back into user code through a delegate. These delegates are usually passed to the framework through a parameter of a method.

Events are a special case of callbacks that supports convenient and consistent syntax for supplying the delegate (an event handler). In addition, Visual Studio's statement completion and designers provide help in using event-based APIs. (See [Event Design](#).)

✓ **CONSIDER** using callbacks to allow users to provide custom code to be executed by the framework.

✓ **CONSIDER** using events to allow users to customize the behavior of a framework without the need for understanding object-oriented design.

✓ **DO** prefer events over plain callbacks, because they are more familiar to a broader range of developers and are integrated with Visual Studio statement completion.

✗ **AVOID** using callbacks in performance-sensitive APIs.

✓ **DO** use the new `Func<...>`, `Action<...>`, or `Expression<...>` types instead of custom delegates, when defining APIs with callbacks.

`Func<...>` and `Action<...>` represent generic delegates. `Expression<...>` represents function definitions that can be compiled and subsequently invoked at runtime but can also be serialized and passed to remote processes.

✓ **DO** measure and understand performance implications of using `Expression<...>`, instead of using `Func<...>` and `Action<...>` delegates.

`Expression<...>` types are in most cases logically equivalent to `Func<...>` and `Action<...>` delegates. The main difference between them is that the delegates are intended to be used in local process scenarios; expressions are intended for cases where it's beneficial and possible to evaluate the expression in a remote process or machine.

✓ **DO** understand that by calling a delegate, you are executing arbitrary code and that could have security, correctness, and compatibility repercussions.

Portions © 2005, 2009 Microsoft Corporation. All rights reserved.

Reprinted by permission of Pearson Education, Inc. from [Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries, 2nd Edition](#) by Krzysztof Cwalina and Brad Abrams, published Oct 22, 2008 by Addison-Wesley Professional as part of the Microsoft Windows Development Series.

See Also

Other Resources

[Designing for Extensibility](#)

[Framework Design Guidelines](#)

Virtual Members

.NET Framework 4.5

Virtual members can be overridden, thus changing the behavior of the subclass. They are quite similar to callbacks in terms of the extensibility they provide, but they are better in terms of execution performance and memory consumption. Also, virtual members feel more natural in scenarios that require creating a special kind of an existing type (specialization).

Virtual members perform better than callbacks and events, but do not perform better than non-virtual methods.

The main disadvantage of virtual members is that the behavior of a virtual member can only be modified at the time of compilation. The behavior of a callback can be modified at runtime.

Virtual members, like callbacks (and maybe more than callbacks), are costly to design, test, and maintain because any call to a virtual member can be overridden in unpredictable ways and can execute arbitrary code. Also, much more effort is usually required to clearly define the contract of virtual members, so the cost of designing and documenting them is higher.

X DO NOT make members virtual unless you have a good reason to do so and you are aware of all the costs related to designing, testing, and maintaining virtual members.

Virtual members are less forgiving in terms of changes that can be made to them without breaking compatibility. Also, they are slower than non-virtual members, mostly because calls to virtual members are not inlined.

✓ **CONSIDER** limiting extensibility to only what is absolutely necessary.

✓ **DO** prefer protected accessibility over public accessibility for virtual members. Public members should provide extensibility (if required) by calling into a protected virtual member.

The public members of a class should provide the right set of functionality for direct consumers of that class. Virtual members are designed to be overridden in subclasses, and protected accessibility is a great way to scope all virtual extensibility points to where they can be used.

Portions © 2005, 2009 Microsoft Corporation. All rights reserved.

Reprinted by permission of Pearson Education, Inc. from [Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries, 2nd Edition](#) by Krzysztof Cwalina and Brad Abrams, published Oct 22, 2008 by Addison-Wesley Professional as part of the Microsoft Windows Development Series.

See Also

Other Resources

[Framework Design Guidelines](#)

[Designing for Extensibility](#)

Abstractions (Abstract Types and Interfaces)

.NET Framework 4.5

An abstraction is a type that describes a contract but does not provide a full implementation of the contract. Abstractions are usually implemented as abstract classes or interfaces, and they come with a well-defined set of reference documentation describing the required semantics of the types implementing the contract. Some of the most important abstractions in the .NET Framework include [Stream](#), [IEnumerable<T>](#), and [Object](#).

You can extend frameworks by implementing a concrete type that supports the contract of an abstraction and using this concrete type with framework APIs consuming (operating on) the abstraction.

A meaningful and useful abstraction that is able to withstand the test of time is very difficult to design. The main difficulty is getting the right set of members, no more and no fewer. If an abstraction has too many members, it becomes difficult or even impossible to implement. If it has too few members for the promised functionality, it becomes useless in many interesting scenarios.

Too many abstractions in a framework also negatively affect usability of the framework. It is often quite difficult to understand an abstraction without understanding how it fits into the larger picture of the concrete implementations and the APIs operating on the abstraction. Also, names of abstractions and their members are necessarily abstract, which often makes them cryptic and unapproachable without first understanding the broader context of their usage.

However, abstractions provide extremely powerful extensibility that the other extensibility mechanisms cannot often match. They are at the core of many architectural patterns, such as plug-ins, inversion of control (IoC), pipelines, and so on. They are also extremely important for testability of frameworks. Good abstractions make it possible to stub out heavy dependencies for the purpose of unit testing. In summary, abstractions are responsible for the sought-after richness of the modern object-oriented frameworks.

X DO NOT provide abstractions unless they are tested by developing several concrete implementations and APIs consuming the abstractions.

✓ DO choose carefully between an abstract class and an interface when designing an abstraction.

✓ CONSIDER providing reference tests for concrete implementations of abstractions. Such tests should allow users to test whether their implementations correctly implement the contract.

Portions © 2005, 2009 Microsoft Corporation. All rights reserved.

Reprinted by permission of Pearson Education, Inc. from [Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries, 2nd Edition](#) by Krzysztof Cwalina and Brad Abrams, published Oct 22, 2008 by Addison-Wesley Professional as part of the Microsoft Windows Development Series.

See Also

Other Resources

[Framework Design Guidelines](#)
[Designing for Extensibility](#)

Base Classes for Implementing Abstractions

.NET Framework 4.5

Strictly speaking, a class becomes a base class when another class is derived from it. For the purpose of this section, however, a base class is a class designed mainly to provide a common abstraction or for other classes to reuse some default implementation through inheritance. Base classes usually sit in the middle of inheritance hierarchies, between an abstraction at the root of a hierarchy and several custom implementations at the bottom.

They serve as implementation helpers for implementing abstractions. For example, one of the Framework's abstractions for ordered collections of items is the `ICollection<T>` interface. Implementing `ICollection<T>` is not trivial, and therefore the Framework provides several base classes, such as `Collection<T>` and `KeyedCollection<TKey, TItem>`, which serve as helpers for implementing custom collections.

Base classes are usually not suited to serve as abstractions by themselves, because they tend to contain too much implementation. For example, the `Collection<T>` base class contains lots of implementation related to the fact that it implements the nongeneric `ICollection` interface (to integrate better with nongeneric collections) and to the fact that it is a collection of items stored in memory in one of its fields.

As previously discussed, base classes can provide invaluable help for users who need to implement abstractions, but at the same time they can be a significant liability. They add surface area and increase the depth of inheritance hierarchies and so conceptually complicate the framework. Therefore, base classes should be used only if they provide significant value to the users of the framework. They should be avoided if they provide value only to the implementers of the framework, in which case delegation to an internal implementation instead of inheritance from a base class should be strongly considered.

✓ **CONSIDER** making base classes abstract even if they don't contain any abstract members. This clearly communicates to the users that the class is designed solely to be inherited from.

✓ **CONSIDER** placing base classes in a separate namespace from the mainline scenario types. By definition, base classes are intended for advanced extensibility scenarios and therefore are not interesting to the majority of users.

✗ **AVOID** naming base classes with a "Base" suffix if the class is intended for use in public APIs.

Portions © 2005, 2009 Microsoft Corporation. All rights reserved.

Reprinted by permission of Pearson Education, Inc. from *Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries, 2nd Edition* by Krzysztof Cwalina and Brad Abrams, published Oct 22, 2008 by Addison-Wesley Professional as part of the Microsoft Windows Development Series.

See Also

Other Resources

[Framework Design Guidelines](#)
[Designing for Extensibility](#)

Sealing

.NET Framework 4.5

One of the features of object-oriented frameworks is that developers can extend and customize them in ways unanticipated by the framework designers. This is both the power and danger of extensible design. When you design your framework, it is, therefore, very important to carefully design for extensibility when it is desired, and to limit extensibility when it is dangerous.

A powerful mechanism that prevents extensibility is sealing. You can seal either the class or individual members. Sealing a class prevents users from inheriting from the class. Sealing a member prevents users from overriding a particular member.

X DO NOT seal classes without having a good reason to do so.

Sealing a class because you cannot think of an extensibility scenario is not a good reason. Framework users like to inherit from classes for various nonobvious reasons, like adding convenience members. See [Unsealed Classes](#) for examples of nonobvious reasons users want to inherit from a type.

Good reasons for sealing a class include the following:

- The class is a static class. See [Static Class Design](#).
- The class stores security-sensitive secrets in inherited protected members.
- The class inherits many virtual members and the cost of sealing them individually would outweigh the benefits of leaving the class unsealed.
- The class is an attribute that requires very fast runtime look-up. Sealed attributes have slightly higher performance levels than unsealed ones. See [Attributes](#).

X DO NOT declare protected or virtual members on sealed types.

By definition, sealed types cannot be inherited from. This means that protected members on sealed types cannot be called, and virtual methods on sealed types cannot be overridden.

✓ **CONSIDER** sealing members that you override.

Problems that can result from introducing virtual members (discussed in [Virtual Members](#)) apply to overrides as well, although to a slightly lesser degree. Sealing an override shields you from these problems starting from that point in the inheritance hierarchy.

Portions © 2005, 2009 Microsoft Corporation. All rights reserved.

Reprinted by permission of Pearson Education, Inc. from [Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries, 2nd Edition](#) by Krzysztof Cwalina and Brad Abrams, published Oct 22, 2008 by Addison-Wesley Professional as part of the Microsoft Windows Development Series.

▲ See Also

Concepts

[Unsealed Classes](#)

Other Resources

[Framework Design Guidelines](#)

[Designing for Extensibility](#)

Design Guidelines for Exceptions

.NET Framework 4.5

Exception handling has many advantages over return-value-based error reporting. Good framework design helps the application developer realize the benefits of exceptions. This section discusses the benefits of exceptions and presents guidelines for using them effectively.

▲ In This Section

[Exception Throwing](#)
[Using Standard Exception Types](#)
[Exceptions and Performance](#)

Portions © 2005, 2009 Microsoft Corporation. All rights reserved.

Reprinted by permission of Pearson Education, Inc. from [Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries, 2nd Edition](#) by Krzysztof Cwalina and Brad Abrams, published Oct 22, 2008 by Addison-Wesley Professional as part of the Microsoft Windows Development Series.

▲ See Also

Other Resources
[Framework Design Guidelines](#)

Exception Throwing

.NET Framework 4.5

Exception-throwing guidelines described in this section require a good definition of the meaning of execution failure. Execution failure occurs whenever a member cannot do what it was designed to do (what the member name implies). For example, if the `OpenFile` method cannot return an opened file handle to the caller, it would be considered an execution failure.

Most developers have become comfortable with using exceptions for usage errors such as division by zero or null references. In the Framework, exceptions are used for all error conditions, including execution errors.

X DO NOT return error codes.

Exceptions are the primary means of reporting errors in frameworks.

✓ **DO** report execution failures by throwing exceptions.

✓ **CONSIDER** terminating the process by calling `System.Environment.FailFast` (.NET Framework 2.0 feature) instead of throwing an exception if your code encounters a situation where it is unsafe for further execution.

X DO NOT use exceptions for the normal flow of control, if possible.

Except for system failures and operations with potential race conditions, framework designers should design APIs so users can write code that does not throw exceptions. For example, you can provide a way to check preconditions before calling a member so users can write code that does not throw exceptions.

The member used to check preconditions of another member is often referred to as a tester, and the member that actually does the work is called a doer.

There are cases when the Tester-Doer Pattern can have an unacceptable performance overhead. In such cases, the so-called Try-Parse Pattern should be considered (see [Exceptions and Performance](#) for more information).

✓ **CONSIDER** the performance implications of throwing exceptions. Throw rates above 100 per second are likely to noticeably impact the performance of most applications.

✓ **DO** document all exceptions thrown by publicly callable members because of a violation of the member contract (rather than a system failure) and treat them as part of your contract.

Exceptions that are a part of the contract should not change from one version to the next (i.e., exception type should not change, and new exceptions should not be added).

X DO NOT have public members that can either throw or not based on some option.

X DO NOT have public members that return exceptions as the return value or an `out` parameter.

Returning exceptions from public APIs instead of throwing them defeats many of the benefits of exception-based error reporting.

✓ **CONSIDER** using exception builder methods.

It is common to throw the same exception from different places. To avoid code bloat, use helper methods that create exceptions and initialize their properties.

Also, members that throw exceptions are not getting inlined. Moving the throw statement inside the builder might allow the member to be inlined.

X DO NOT throw exceptions from exception filter blocks.

When an exception filter raises an exception, the exception is caught by the CLR, and the filter returns false. This behavior is indistinguishable from the filter executing and returning false explicitly and is therefore very difficult to debug.

X AVOID explicitly throwing exceptions from finally blocks. Implicitly thrown exceptions resulting from calling methods that throw are acceptable.

Portions © 2005, 2009 Microsoft Corporation. All rights reserved.

Reprinted by permission of Pearson Education, Inc. from [Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries, 2nd Edition](#) by Krzysztof Cwalina and Brad Abrams, published Oct 22, 2008 by Addison-Wesley Professional as part of the Microsoft Windows Development Series.

▲ See Also

Other Resources

[Framework Design Guidelines](#)

[Design Guidelines for Exceptions](#)

Using Standard Exception Types

.NET Framework 4.5

This section describes the standard exceptions provided by the Framework and the details of their usage. The list is by no means exhaustive. Please refer to the .NET Framework reference documentation for usage of other Framework exception types.

▲ Exception and SystemException

- X **DO NOT** throw [System.Exception](#) or [System.SystemException](#).
- X **DO NOT** catch [System.Exception](#) or [System.SystemException](#) in framework code, unless you intend to rethrow.
- X **AVOID** catching [System.Exception](#) or [System.SystemException](#), except in top-level exception handlers.

▲ ApplicationException

- X **DO NOT** throw or derive from [ApplicationException](#).

▲ InvalidOperationException

- ✓ **DO** throw an [InvalidOperationException](#) if the object is in an inappropriate state.

▲ ArgumentException, ArgumentNullException, and ArgumentOutOfRangeException

- ✓ **DO** throw [ArgumentException](#) or one of its subtypes if bad arguments are passed to a member. Prefer the most derived exception type, if applicable.
 - ✓ **DO** set the [ParamName](#) property when throwing one of the subclasses of [ArgumentExcepti on](#).
- This property represents the name of the parameter that caused the exception to be thrown. Note that the property can be set using one of the constructor overloads.
- ✓ **DO** use [value](#) for the name of the implicit value parameter of property setters.

▲ NullReferenceException, IndexOutOfRangeException, and AccessViolationException

- X **DO NOT** allow publicly callable APIs to explicitly or implicitly throw [NullReferenceException](#), [AccessViolationException](#), or [IndexOutOfRangeException](#). These exceptions are reserved and thrown by the execution engine and in most cases indicate a bug.
- Do argument checking to avoid throwing these exceptions. Throwing these exceptions exposes implementation details of your method that might change over time.

▲ StackOverflowException

- X **DO NOT** explicitly throw [StackOverflowException](#). The exception should be explicitly thrown only by the CLR.
 - X **DO NOT** catch [StackOverfl owExcepti on](#).
- It is almost impossible to write managed code that remains consistent in the presence of arbitrary stack overflows. The unmanaged parts of the CLR remain consistent by using probes to move stack overflows to well-defined places rather than by backing out from arbitrary stack overflows.

▲ OutOfMemoryException

- X **DO NOT** explicitly throw [OutOfMemoryException](#). This exception is to be thrown only by the CLR infrastructure.

▲ ComException, SEHException, and ExecutionEngineException

- X **DO NOT** explicitly throw [COMException](#), [ExecutionEngineException](#), and [SEHException](#). These exceptions are to be thrown only by the CLR infrastructure.

Portions © 2005, 2009 Microsoft Corporation. All rights reserved.

Reprinted by permission of Pearson Education, Inc. from *Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries, 2nd Edition* by Krzysztof Cwalina and Brad Abrams, published Oct 22, 2008 by Addison-Wesley Professional as part of the Microsoft Windows Development Series.

▲ See Also

Other Resources

[Framework Design Guidelines](#)

[Design Guidelines for Exceptions](#)

Exceptions and Performance

.NET Framework 4.5

One common concern related to exceptions is that if exceptions are used for code that routinely fails, the performance of the implementation will be unacceptable. This is a valid concern. When a member throws an exception, its performance can be orders of magnitude slower. However, it is possible to achieve good performance while strictly adhering to the exception guidelines that disallow using error codes. Two patterns described in this section suggest ways to do this.

X DO NOT use error codes because of concerns that exceptions might affect performance negatively.

To improve performance, it is possible to use either the Tester-Doer Pattern or the Try-Parse Pattern, described in the next two sections.

Tester-Doer Pattern

Sometimes performance of an exception-throwing member can be improved by breaking the member into two. Let's look at the [Add](#) method of the [ICollection<T>](#) interface.

```
ICollection<int> numbers = ...
numbers.Add(1);
```

The method [Add](#) throws if the collection is read-only. This can be a performance problem in scenarios where the method call is expected to fail often. One of the ways to mitigate the problem is to test whether the collection is writable before trying to add a value.

```
ICollection<int> numbers = ...
...
if(!numbers.IsReadOnly){
    numbers.Add(1);
}
```

The member used to test a condition, which in our example is the property [IsReadOnly](#), is referred to as the tester. The member used to perform a potentially throwing operation, the [Add](#) method in our example, is referred to as the doer.

✓ **CONSIDER** the Tester-Doer Pattern for members that might throw exceptions in common scenarios to avoid performance problems related to exceptions.

Try-Parse Pattern

For extremely performance-sensitive APIs, an even faster pattern than the Tester-Doer Pattern described in the previous section should be used. The pattern calls for adjusting the member name to make a well-defined test case a part of the member semantics. For example, [DateTime](#) defines a [Parse](#) method that throws an exception if parsing of a string fails. It also defines a corresponding [TryParse](#) method that attempts to parse, but returns false if parsing is unsuccessful and returns the result of a successful parsing using an [out](#) parameter.

```
public struct DateTime {
    public static DateTime Parse(string dateTime){
        ...
    }
    public static bool TryParse(string dateTime, out DateTime result){
        ...
    }
}
```

When using this pattern, it is important to define the try functionality in strict terms. If the member fails for any reason other than the well-defined try, the member must still throw a corresponding exception.

✓ **CONSIDER** the Try-Parse Pattern for members that might throw exceptions in common scenarios to avoid performance problems related to exceptions.

✓ **DO** use the prefix "Try" and Boolean return type for methods implementing this pattern.

✓ **DO** provide an exception-throwing member for each member using the Try-Parse Pattern.

Portions © 2005, 2009 Microsoft Corporation. All rights reserved.

Reprinted by permission of Pearson Education, Inc. from [Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries, 2nd Edition](#) by Krzysztof Cwalina and Brad Abrams, published Oct 22, 2008 by Addison-Wesley Professional as part of the Microsoft Windows Development Series.

See Also

Other Resources

[Framework Design Guidelines](#)

Usage Guidelines

.NET Framework 4.5

This section contains guidelines for using common types in publicly accessible APIs. It deals with direct usage of built-in Framework types (e.g., serialization attributes) and overloading common operators.

The [System.IDisposable](#) interface is not covered in this section, but is discussed in the [Dispose Pattern](#) section.

Note

For guidelines and additional information about other common, built-in .NET Framework types, see the reference topics for the following: [System.DateTime](#), [System.DateTimeOffset](#), [System.ICloneable](#), [System.IComparable<T>](#), [System.IEquatable<T>](#), [System.Nullable<T>](#), [System.Object](#), [System.Uri](#).

▲ In This Section

- [Arrays](#)
- [Attributes](#)
- [Collections](#)
- [Serialization](#)
- [System.Xml Usage](#)
- [Equality Operators](#)

Portions © 2005, 2009 Microsoft Corporation. All rights reserved.

Reprinted by permission of Pearson Education, Inc. from [Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries, 2nd Edition](#) by Krzysztof Cwalina and Brad Abrams, published Oct 22, 2008 by Addison-Wesley Professional as part of the Microsoft Windows Development Series.

▲ See Also

- [Other Resources](#)
- [Framework Design Guidelines](#)

Arrays

.NET Framework 4.5

✓ **DO** prefer using collections over arrays in public APIs. The [Guidelines for Collections](#) section provides details about how to choose between collections and arrays.

X **DO NOT** use read-only array fields. The field itself is read-only and can't be changed, but elements in the array can be changed.

✓ **CONSIDER** using jagged arrays instead of multidimensional arrays.

A jagged array is an array with elements that are also arrays. The arrays that make up the elements can be of different sizes, leading to less wasted space for some sets of data (e.g., sparse matrix) compared to multidimensional arrays. Furthermore, the CLR optimizes index operations on jagged arrays, so they might exhibit better runtime performance in some scenarios.

Portions © 2005, 2009 Microsoft Corporation. All rights reserved.

Reprinted by permission of Pearson Education, Inc. from [Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries, 2nd Edition](#) by Krzysztof Cwalina and Brad Abrams, published Oct 22, 2008 by Addison-Wesley Professional as part of the Microsoft Windows Development Series.

▲ See Also

Reference

[Array](#)

Other Resources

[Framework Design Guidelines](#)

[Usage Guidelines](#)

Attributes

.NET Framework 4.5

[System.Attribute](#) is a base class used to define custom attributes.

Attributes are annotations that can be added to programming elements such as assemblies, types, members, and parameters. They are stored in the metadata of the assembly and can be accessed at runtime using the reflection APIs. For example, the Framework defines the [ObsoleteAttribute](#), which can be applied to a type or a member to indicate that the type or member has been deprecated.

Attributes can have one or more properties that carry additional data related to the attribute. For example, [ObsoleteAttribute](#) could carry additional information about the release in which a type or a member got deprecated and the description of the new APIs replacing the obsolete API.

Some properties of an attribute must be specified when the attribute is applied. These are referred to as the required properties or required arguments, because they are represented as positional constructor parameters. For example, the [ConditionString](#) property of the [ConditionalAttribute](#) is a required property.

Properties that do not necessarily have to be specified when the attribute is applied are called optional properties (or optional arguments). They are represented by settable properties. Compilers provide special syntax to set these properties when an attribute is applied. For example, the [AttributeUsageAttribute.Inherited](#) property represents an optional argument.

✓ **DO** name custom attribute classes with the suffix "Attribute."

✓ **DO** apply the [AttributeUsageAttribute](#) to custom attributes.

✓ **DO** provide settable properties for optional arguments.

✓ **DO** provide get-only properties for required arguments.

✓ **DO** provide constructor parameters to initialize properties corresponding to required arguments. Each parameter should have the same name (although with different casing) as the corresponding property.

✗ **AVOID** providing constructor parameters to initialize properties corresponding to the optional arguments.

In other words, do not have properties that can be set with both a constructor and a setter. This guideline makes very explicit which arguments are optional and which are required, and avoids having two ways of doing the same thing.

✗ **AVOID** overloading custom attribute constructors.

Having only one constructor clearly communicates to the user which arguments are required and which are optional.

✓ **DO** seal custom attribute classes, if possible. This makes the look-up for the attribute faster.

Portions © 2005, 2009 Microsoft Corporation. All rights reserved.

Reprinted by permission of Pearson Education, Inc. from [Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries, 2nd Edition](#) by Krzysztof Cwalina and Brad Abrams, published Oct 22, 2008 by Addison-Wesley Professional as part of the Microsoft Windows Development Series.

See Also

Other Resources

[Framework Design Guidelines](#)

[Usage Guidelines](#)

Guidelines for Collections

.NET Framework 4.5

Any type designed specifically to manipulate a group of objects having some common characteristic can be considered a collection. It is almost always appropriate for such types to implement `IEnumerable` or `IEnumerable<T>`, so in this section we only consider types implementing one or both of those interfaces to be collections.

X DO NOT use weakly typed collections in public APIs.

The type of all return values and parameters representing collection items should be the exact item type, not any of its base types (this applies only to public members of the collection).

X DO NOT use `ArrayList` or `List<T>` in public APIs.

These types are data structures designed to be used in internal implementation, not in public APIs. `List<T>` is optimized for performance and power at the cost of cleanness of the APIs and flexibility. For example, if you return `List<T>`, you will not ever be able to receive notifications when client code modifies the collection. Also, `List<T>` exposes many members, such as `BinarySearch`, that are not useful or applicable in many scenarios. The following two sections describe types (abstractions) designed specifically for use in public APIs.

X DO NOT use `Hashtable` or `Dictionary<TKey, TValue>` in public APIs.

These types are data structures designed to be used in internal implementation. Public APIs should use `IDictionary`, `IDictionary<TKey, TValue>`, or a custom type implementing one or both of the interfaces.

X DO NOT use `IEnumerator<T>`, `IEnumerator`, or any other type that implements either of these interfaces, except as the return type of a `GetEnumerator` method.

Types returning enumerators from methods other than `GetEnumerator` cannot be used with the `foreach` statement.

X DO NOT implement both `IEnumerator<T>` and `IEnumerable<T>` on the same type. The same applies to the nongeneric interfaces `IEnumerator` and `IEnumerable`.

Collection Parameters

✓ **DO** use the least-specialized type possible as a parameter type. Most members taking collections as parameters use the `IEnumerable<T>` interface.

X AVOID using `ICollection<T>` or `ICollection` as a parameter just to access the `Count` property.

Instead, consider using `IEnumerable<T>` or `IEnumerable` and dynamically checking whether the object implements `ICollection<T>` or `ICollection`.

Collection Properties and Return Values

X DO NOT provide settable collection properties.

Users can replace the contents of the collection by clearing the collection first and then adding the new contents. If replacing the whole collection is a common scenario, consider providing the `AddRange` method on the collection.

✓ **DO** use `ICollection<T>` or a subclass of `ICollection<T>` for properties or return values representing read/write collections.

If `ICollection<T>` does not meet some requirement (e.g., the collection must not implement `IList`), use a custom collection by implementing `IEnumerable<T>`, `ICollection<T>`, or `IList<T>`.

✓ **DO** use `ReadOnlyCollection<T>`, a subclass of `ReadOnlyCollection<T>`, or in rare cases `IEnumerable<T>` for properties or return values representing read-only collections.

In general, prefer `ReadOnlyCollection<T>`. If it does not meet some requirement (e.g., the collection must not implement `IList`), use a custom collection by implementing `IEnumerable<T>`, `ICollection<T>`, or `IList<T>`. If you do implement a custom read-only collection, implement `ICollection<T>`. `ReadOnly` to return false.

In cases where you are sure that the only scenario you will ever want to support is forward-only iteration, you can simply use `IEnumerable<T>`.

✓ **CONSIDER** using subclasses of generic base collections instead of using the collections directly.

This allows for a better name and for adding helper members that are not present on the base collection types. This is especially applicable to high-level APIs.

✓ **CONSIDER** returning a subclass of `ICollection<T>` or `ReadOnlyCollection<T>` from very commonly used methods and properties.

This will make it possible for you to add helper methods or change the collection implementation in the future.

✓ **CONSIDER** using a keyed collection if the items stored in the collection have unique keys (names, IDs, etc.). Keyed collections are collections that can be indexed by both an integer and a key and are usually implemented by inheriting from `KeyedCollection<TKey, TItem>`.

Keyed collections usually have larger memory footprints and should not be used if the memory overhead outweighs the benefits of having the keys.

X DO NOT return null values from collection properties or from methods returning collections. Return an empty collection or an empty array instead.

The general rule is that null and empty (0 item) collections or arrays should be treated the same.

Snapshots Versus Live Collections

Collections representing a state at some point in time are called snapshot collections. For example, a collection containing rows returned from a database query

would be a snapshot. Collections that always represent the current state are called live collections. For example, a collection of `ComboBox` items is a live collection.

X DO NOT return snapshot collections from properties. Properties should return live collections.

Property getters should be very lightweight operations. Returning a snapshot requires creating a copy of an internal collection in an $O(n)$ operation.

✓ **DO** use either a snapshot collection or a live `IEnumerable<T>` (or its subtype) to represent collections that are volatile (i.e., that can change without explicitly modifying the collection).

In general, all collections representing a shared resource (e.g., files in a directory) are volatile. Such collections are very difficult or impossible to implement as live collections unless the implementation is simply a forward-only enumerator.

Choosing Between Arrays and Collections

✓ **DO** prefer collections over arrays.

Collections provide more control over contents, can evolve over time, and are more usable. In addition, using arrays for read-only scenarios is discouraged because the cost of cloning the array is prohibitive. Usability studies have shown that some developers feel more comfortable using collection-based APIs.

However, if you are developing low-level APIs, it might be better to use arrays for read-write scenarios. Arrays have a smaller memory footprint, which helps reduce the working set, and access to elements in an array is faster because it is optimized by the runtime.

✓ **CONSIDER** using arrays in low-level APIs to minimize memory consumption and maximize performance.

✓ **DO** use byte arrays instead of collections of bytes.

X DO NOT use arrays for properties if the property would have to return a new array (e.g., a copy of an internal array) every time the property getter is called.

Implementing Custom Collections

✓ **CONSIDER** inheriting from `Collection<T>`, `ReadOnlyCollection<T>`, or `KeyedCollection<TKey, TItem>` when designing new collections.

✓ **DO** implement `IEnumerable<T>` when designing new collections. Consider implementing `ICollection<T>` or even `IList<T>` where it makes sense.

When implementing such custom collection, follow the API pattern established by `Collection<T>` and `ReadOnlyCollection<T>` as closely as possible. That is, implement the same members explicitly, name the parameters like these two collections name them, and so on.

✓ **CONSIDER** implementing nongeneric collection interfaces (`IList` and `ICollection`) if the collection will often be passed to APIs taking these interfaces as input.

X AVOID implementing collection interfaces on types with complex APIs unrelated to the concept of a collection.

X DO NOT inherit from nongeneric base collections such as `CollectionBase`. Use `Collection<T>`, `ReadOnlyCollection<T>`, and `KeyedCollection<TKey, TItem>` instead.

Naming Custom Collections

Collections (types that implement `IEnumerable`) are created mainly for two reasons: (1) to create a new data structure with structure-specific operations and often different performance characteristics than existing data structures (e.g., `List<T>`, `LinkedList<T>`, `Stack<T>`), and (2) to create a specialized collection for holding a specific set of items (e.g., `StringCollection`). Data structures are most often used in the internal implementation of applications and libraries. Specialized collections are mainly to be exposed in APIs (as property and parameter types).

✓ **DO** use the "Dictionary" suffix in names of abstractions implementing `IDictionary` or `IDictionary<TKey, TValue>`.

✓ **DO** use the "Collection" suffix in names of types implementing `IEnumerable` (or any of its descendants) and representing a list of items.

✓ **DO** use the appropriate data structure name for custom data structures.

X AVOID using any suffixes implying particular implementation, such as "LinkedList" or "Hashtable," in names of collection abstractions.

✓ **CONSIDER** prefixing collection names with the name of the item type. For example, a collection storing items of type `Address` (implementing `IEnumerable<Address>`) should be named `AddressCollection`. If the item type is an interface, the "I" prefix of the item type can be omitted. Thus, a collection of `IDisposable` items can be called `DisposableCollection`.

✓ **CONSIDER** using the "ReadOnly" prefix in names of read-only collections if a corresponding writeable collection might be added or already exists in the framework.

For example, a read-only collection of strings should be called `ReadOnlyStringCollection`.

Portions © 2005, 2009 Microsoft Corporation. All rights reserved.

Reprinted by permission of Pearson Education, Inc. from [Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries, 2nd Edition](#) by Krzysztof Cwalina and Brad Abrams, published Oct 22, 2008 by Addison-Wesley Professional as part of the Microsoft Windows Development Series.

See Also

Other Resources
[Framework Design Guidelines](#)

Serialization

.NET Framework 4.5

Serialization is the process of converting an object into a format that can be readily persisted or transported. For example, you can serialize an object, transport it over the Internet using HTTP, and deserialize it at the destination machine.

The .NET Framework offers three main serialization technologies optimized for various serialization scenarios. The following table lists these technologies and the main Framework types related to these technologies.

Technology Name	Main Types	Scenarios
Data Contract Serialization	DataContractAttribute DataMemberAttribute DataContractSerializer NetDataContractSerializer DataContractJsonSerializer ISerializable	General persistence Web Services JSON
XML Serialization	XmlSerializer	XML format with full control over the shape of the XML
Runtime Serialization (Binary and SOAP)	SerializableAttribute ISerializable BinaryFormatter SoapFormatter	.NET Remoting

✓ **DO** think about serialization when you design new types.

▀ Choosing the Right Serialization Technology to Support

✓ **CONSIDER** supporting Data Contract Serialization if instances of your type might need to be persisted or used in Web Services.

✓ **CONSIDER** supporting the XML Serialization instead of or in addition to Data Contract Serialization if you need more control over the XML format that is produced when the type is serialized.

This may be necessary in some interoperability scenarios where you need to use an XML construct that is not supported by Data Contract Serialization, for example, to produce XML attributes.

✓ **CONSIDER** supporting the Runtime Serialization if instances of your type need to travel across .NET Remoting boundaries.

✗ **AVOID** supporting Runtime Serialization or XML Serialization just for general persistence reasons. Prefer Data Contract Serialization instead.

▀ Supporting Data Contract Serialization

Types can support Data Contract Serialization by applying the [DataContractAttribute](#) to the type and the [DataMemberAttribute](#) to the members (fields and properties) of the type.

✓ **CONSIDER** marking data members of your type public if the type can be used in partial trust.

In full trust, Data Contract serializers can serialize and deserialize nonpublic types and members, but only public members can be serialized and deserialized in partial trust.

✓ **DO** implement a getter and setter on all properties that have [DataMemberAttribute](#). Data Contract serializers require both the getter and the setter for the type to be considered serializable. (In .NET Framework 3.5 SP1, some collection properties can be get-only.) If the type won't be used in partial trust, one or both of the property accessors can be nonpublic.

✓ **CONSIDER** using the serialization callbacks for initialization of deserialized instances.

Constructors are not called when objects are deserialized. (There are exceptions to the rule. Constructors of collections marked with [CollectionDataContractAttribute](#) are called during deserialization.) Therefore, any logic that executes during normal construction needs to be implemented as one of the serialization callbacks.

[OnDeserializingAttribute](#) is the most commonly used callback attribute. The other attributes in the family are [OnDeserializingAttribute](#), [OnSerializingAttribute](#), and [OnSerializedAttribute](#). They can be used to mark callbacks that get executed before deserialization, before serialization, and finally, after serialization, respectively.

✓ **CONSIDER** using the [KnownTypeAttribute](#) to indicate concrete types that should be used when deserializing a complex object graph.

✓ **DO** consider backward and forward compatibility when creating or changing serializable types.

Keep in mind that serialized streams of future versions of your type can be deserialized into the current version of the type, and vice versa.

Make sure you understand that data members, even private and internal, cannot change their names, types, or even their order in future versions of the type unless special care is taken to preserve the contract using explicit parameters to the data contract attributes.

Test compatibility of serialization when making changes to serializable types. Try deserializing the new version into an old version, and vice versa.

✓ **CONSIDER** implementing [IExtensibleDataObject](#) to allow round-tripping between different versions of the type.

The interface allows the serializer to ensure that no data is lost during round-tripping. The [IExtensibleDataObject.ExtensionData](#) property is used to store any data from

the future version of the type that is unknown to the current version, and so it cannot store it in its data members. When the current version is subsequently serialized and deserialized into a future version, the additional data will be available in the serialized stream.

Supporting XML Serialization

Data Contract Serialization is the main (default) serialization technology in the .NET Framework, but there are serialization scenarios that Data Contract Serialization does not support. For example, it does not give you full control over the shape of XML produced or consumed by the serializer. If such fine control is required, XML Serialization has to be used, and you need to design your types to support this serialization technology.

X AVOID designing your types specifically for XML Serialization, unless you have a very strong reason to control the shape of the XML produced. This serialization technology has been superseded by the Data Contract Serialization discussed in the previous section.

✓ CONSIDER implementing the [IXmlSerializable](#) interface if you want even more control over the shape of the serialized XML than what's offered by applying the XML Serialization attributes. Two methods of the interface, [ReadXml](#) and [WriteXml](#), allow you to fully control the serialized XML stream. You can also control the XML schema that gets generated for the type by applying the [Xml SchemaProviderAttribute](#).

Supporting Runtime Serialization

Runtime Serialization is a technology used by .NET Remoting. If you think your types will be transported using .NET Remoting, you need to make sure they support Runtime Serialization.

The basic support for Runtime Serialization can be provided by applying the [SerializableAttribute](#), and more advanced scenarios involve implementing a simple Runtime Serializable Pattern (implement [ISerializable](#) and provide serialization constructor).

✓ CONSIDER supporting Runtime Serialization if your types will be used with .NET Remoting. For example, the [System.AddIn](#) namespace uses .NET Remoting, and so all types exchanged between [System.AddIn](#) add-ins need to support Runtime Serialization.

✓ CONSIDER implementing the Runtime Serializable Pattern if you want complete control over the serialization process. For example, if you want to transform data as it gets serialized or deserialized.

The pattern is very simple. All you need to do is implement the [ISerializable](#) interface and provide a special constructor that is used when the object is deserialized.

✓ DO make the serialization constructor protected and provide two parameters typed and named exactly as shown in the sample here.

```
[Serializable]
public class Person : ISerializable {
    protected Person(SerializationInfo info, StreamingContext context) {
        ...
    }
}
```

✓ DO implement the [ISerializable](#) members explicitly.

✓ DO apply a link demand to [ISerializable.GetObjectData](#) implementation. This ensures that only fully trusted code and the Runtime Serializer have access to the member.

Portions © 2005, 2009 Microsoft Corporation. All rights reserved.

Reprinted by permission of Pearson Education, Inc. from [Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries, 2nd Edition](#) by Krzysztof Cwalina and Brad Abrams, published Oct 22, 2008 by Addison-Wesley Professional as part of the Microsoft Windows Development Series.

See Also

Other Resources
[Framework Design Guidelines](#)
[Usage Guidelines](#)

System.Xml Usage

.NET Framework 4.5

This section talks about usage of several types residing in [System.Xml](#) namespaces that can be used to represent XML data.

X DO NOT use [XmlNode](#) or [XmlDocument](#) to represent XML data. Favor using instances of [IXPathNavigable](#), [XmlReader](#), [XmlWriter](#), or subtypes of [XmlNode](#) instead. [XmlNode](#) and [XmlDocument](#) are not designed for exposing in public APIs.

✓ **DO** use [XmlReader](#), [IXPathNavigable](#), or subtypes of [XmlNode](#) as input or output of members that accept or return XML.

Use these abstractions instead of [XmlDocument](#), [XmlNode](#), or [XPathDocument](#), because this decouples the methods from specific implementations of an in-memory XML document and allows them to work with virtual XML data sources that expose [XmlNode](#), [XmlReader](#), or [XPathNavigator](#).

X DO NOT subclass [XmlDocument](#) if you want to create a type representing an XML view of an underlying object model or data source.

Portions © 2005, 2009 Microsoft Corporation. All rights reserved.

Reprinted by permission of Pearson Education, Inc. from [Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries, 2nd Edition](#) by Krzysztof Cwalina and Brad Abrams, published Oct 22, 2008 by Addison-Wesley Professional as part of the Microsoft Windows Development Series.

▲ See Also

Other Resources
[Framework Design Guidelines](#)
[Usage Guidelines](#)

Equality Operators

.NET Framework 4.5

This section discusses overloading equality operators and refers to `operator==` and `operator!=` as equality operators.

X DO NOT overload one of the equality operators and not the other.

✓ **DO** ensure that `Object.Equals` and the equality operators have exactly the same semantics and similar performance characteristics.

This often means that `Object.Equals` needs to be overridden when the equality operators are overloaded.

X AVOID throwing exceptions from equality operators.

For example, return false if one of the arguments is null instead of throwing `NullReferenceException`.

Equality Operators on Value Types

✓ **DO** overload the equality operators on value types, if equality is meaningful.

In most programming languages, there is no default implementation of `operator==` for value types.

Equality Operators on Reference Types

X AVOID overloading equality operators on mutable reference types.

Many languages have built-in equality operators for reference types. The built-in operators usually implement the reference equality, and many developers are surprised when the default behavior is changed to the value equality.

This problem is mitigated for immutable reference types because immutability makes it much harder to notice the difference between reference equality and value equality.

X AVOID overloading equality operators on reference types if the implementation would be significantly slower than that of reference equality.

Portions © 2005, 2009 Microsoft Corporation. All rights reserved.

Reprinted by permission of Pearson Education, Inc. from [Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries, 2nd Edition](#) by Krzysztof Walina and Brad Abrams, published Oct 22, 2008 by Addison-Wesley Professional as part of the Microsoft Windows Development Series.

See Also

Other Resources
[Framework Design Guidelines](#)
[Usage Guidelines](#)

Common Design Patterns

.NET Framework 4.5

There are numerous books on software patterns, pattern languages, and antipatterns that address the very broad subject of patterns. Thus, this chapter provides guidelines and discussion related to a very limited set of patterns that are used frequently in the design of the .NET Framework APIs.

▲ In This Section

[Dependency Properties](#)
[Dispose Pattern](#)

Portions © 2005, 2009 Microsoft Corporation. All rights reserved.

Reprinted by permission of Pearson Education, Inc. from [Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries, 2nd Edition](#) by Krzysztof Cwalina and Brad Abrams, published Oct 22, 2008 by Addison-Wesley Professional as part of the Microsoft Windows Development Series.

▲ See Also

[Other Resources](#)
[Framework Design Guidelines](#)

Dependency Properties

.NET Framework 4.5

A dependency property (DP) is a regular property that stores its value in a property store instead of storing it in a type variable (field), for example.

An attached dependency property is a kind of dependency property modeled as static Get and Set methods representing "properties" describing relationships between objects and their containers (e.g., the position of a `Button` object on a `Panel` container).

✓ **DO** provide the dependency properties, if you need the properties to support WPF features such as styling, triggers, data binding, animations, dynamic resources, and inheritance.

Dependency Property Design

✓ **DO** inherit from `DependencyObject`, or one of its subtypes, when implementing dependency properties. The type provides a very efficient implementation of a property store and automatically supports WPF data binding.

✓ **DO** provide a regular CLR property and public static read-only field storing an instance of `System.Windows.DependencyProperty` for each dependency property.

✓ **DO** implement dependency properties by calling instance methods `DependencyObject.GetValue` and `DependencyObject.SetValue`.

✓ **DO** name the dependency property static field by suffixing the name of the property with "Property."

X DO NOT set default values of dependency properties explicitly in code; set them in metadata instead.

If you set a property default explicitly, you might prevent that property from being set by some implicit means, such as a styling.

X DO NOT put code in the property accessors other than the standard code to access the static field.

That code won't execute if the property is set by implicit means, such as a styling, because styling uses the static field directly.

X DO NOT use dependency properties to store secure data. Even private dependency properties can be accessed publicly.

Attached Dependency Property Design

Dependency properties described in the preceding section represent intrinsic properties of the declaring type; for example, the `Text` property is a property of `TextButton`, which declares it. A special kind of dependency property is the attached dependency property.

A classic example of an attached property is the `Grid.Column` property. The property represents Button's (not Grid's) column position, but it is only relevant if the Button is contained in a Grid, and so it's "attached" to Buttons by Grids.

```
<Grid>
  <Grid.ColumnDefinitions>
    <ColumnDefinition />
    <ColumnDefinition />
  </Grid.ColumnDefinitions>

  <Button Grid.Column="0">Click</Button>
  <Button Grid.Column="1">Click</Button>
</Grid>
```

The definition of an attached property looks mostly like that of a regular dependency property, except that the accessors are represented by static Get and Set methods:

```
public class Grid {

    public static int GetColumn(DependencyObject obj) {
        return (int)obj.GetValue(ColumnProperty);
    }

    public static void SetColumn(DependencyObject obj, int value) {
        obj.SetValue(ColumnProperty, value);
    }

    public static readonly DependencyProperty ColumnProperty =
        DependencyProperty.RegisterAttached(
            "Column",
            typeof(int),
            typeof(Grid)
        );
}
```

Dependency Property Validation

Properties often implement what is called validation. Validation logic executes when an attempt is made to change the value of a property.

Unfortunately dependency property accessors cannot contain arbitrary validation code. Instead, dependency property validation logic needs to be specified during property registration.

X DO NOT put dependency property validation logic in the property's accessors. Instead, pass a validation callback to [DependencyProperty.Register](#) method.

▲ Dependency Property Change Notifications

X DO NOT implement change notification logic in dependency property accessors. Dependency properties have a built-in change notifications feature that must be used by supplying a change notification callback to the [PropertyMetadata](#).

▲ Dependency Property Value Coercion

Property coercion takes place when the value given to a property setter is modified by the setter before the property store is actually modified.

X DO NOT implement coercion logic in dependency property accessors.

Dependency properties have a built-in coercion feature, and it can be used by supplying a coercion callback to the [PropertyMetadata](#).

Portions © 2005, 2009 Microsoft Corporation. All rights reserved.

Reprinted by permission of Pearson Education, Inc. from [Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries, 2nd Edition](#) by Krzysztof Cwalina and Brad Abrams, published Oct 22, 2008 by Addison-Wesley Professional as part of the Microsoft Windows Development Series.

▲ See Also

Other Resources

[Framework Design Guidelines](#)

[Common Design Patterns](#)

Dispose Pattern

.NET Framework 4.5

All programs acquire one or more system resources, such as memory, system handles, or database connections, during the course of their execution. Developers have to be careful when using such system resources, because they must be released after they have been acquired and used.

The CLR provides support for automatic memory management. Managed memory (memory allocated using the C# operator **new**) does not need to be explicitly released. It is released automatically by the garbage collector (GC). This frees developers from the tedious and difficult task of releasing memory and has been one of the main reasons for the unprecedented productivity afforded by the .NET Framework.

Unfortunately, managed memory is just one of many types of system resources. Resources other than managed memory still need to be released explicitly and are referred to as unmanaged resources. The GC was specifically not designed to manage such unmanaged resources, which means that the responsibility for managing unmanaged resources lies in the hands of the developers.

The CLR provides some help in releasing unmanaged resources. [System.Object](#) declares a virtual method [Finalize](#) (also called the finalizer) that is called by the GC before the object's memory is reclaimed by the GC and can be overridden to release unmanaged resources. Types that override the finalizer are referred to as finalizable types.

Although finalizers are effective in some cleanup scenarios, they have two significant drawbacks:

- The finalizer is called when the GC detects that an object is eligible for collection. This happens at some undetermined period of time after the resource is not needed anymore. The delay between when the developer could or would like to release the resource and the time when the resource is actually released by the finalizer might be unacceptable in programs that acquire many scarce resources (resources that can be easily exhausted) or in cases in which resources are costly to keep in use (e.g., large unmanaged memory buffers).
- When the CLR needs to call a finalizer, it must postpone collection of the object's memory until the next round of garbage collection (the finalizers run between collections). This means that the object's memory (and all objects it refers to) will not be released for a longer period of time.

Therefore, relying exclusively on finalizers might not be appropriate in many scenarios when it is important to reclaim unmanaged resources as quickly as possible, when dealing with scarce resources, or in highly performant scenarios in which the added GC overhead of finalization is unacceptable.

The Framework provides the [System.IDisposable](#) interface that should be implemented to provide the developer a manual way to release unmanaged resources as soon as they are not needed. It also provides the [GC.SuppressFinalize](#) method that can tell the GC that an object was manually disposed of and does not need to be finalized anymore, in which case the object's memory can be reclaimed earlier. Types that implement the [IDisposable](#) interface are referred to as disposable types.

The Dispose Pattern is intended to standardize the usage and implementation of finalizers and the [IDisposable](#) interface.

The main motivation for the pattern is to reduce the complexity of the implementation of the [Finalize](#) and the [Dispose](#) methods. The complexity stems from the fact that the methods share some but not all code paths (the differences are described later in the chapter). In addition, there are historical reasons for some elements of the pattern related to the evolution of language support for deterministic resource management.

✓ **DO** implement the Basic Dispose Pattern on types containing instances of disposable types. See the [Basic Dispose Pattern](#) section for details on the basic pattern.

If a type is responsible for the lifetime of other disposable objects, developers need a way to dispose of them, too. Using the container's [Dispose](#) method is a convenient way to make this possible.

✓ **DO** implement the Basic Dispose Pattern and provide a finalizer on types holding resources that need to be freed explicitly and that do not have finalizers.

For example, the pattern should be implemented on types storing unmanaged memory buffers. The [Finalizable Types](#) section discusses guidelines related to implementing finalizers.

✓ **CONSIDER** implementing the Basic Dispose Pattern on classes that themselves don't hold unmanaged resources or disposable objects but are likely to have subtypes that do.

A great example of this is the [System.IO.Stream](#) class. Although it is an abstract base class that doesn't hold any resources, most of its subclasses do and because of this, it implements this pattern.

Basic Dispose Pattern

The basic implementation of the pattern involves implementing the [System.IDisposable](#) interface and declaring the [Dispose\(bool\)](#) method that implements all resource cleanup logic to be shared between the [Dispose](#) method and the optional finalizer.

The following example shows a simple implementation of the basic pattern:

```
public class DisposableResourceHolder : IDisposable {  
  
    private SafeHandle resource; // handle to a resource  
  
    public DisposableResourceHolder(){  
        this.resource = ... // allocates the resource  
    }  
  
    public void Dispose(){  
        Dispose(true);  
        GC.SuppressFinalize(this);  
    }  
  
    protected virtual void Dispose(bool disposing){  
        if (disposing){
```

```

        }
        if (resource != null) resource.Dispose();
    }
}

```

The Boolean parameter *disposing* indicates whether the method was invoked from the `IDisposable.Dispose` implementation or from the finalizer. The `Dispose(bool)` implementation should check the parameter before accessing other reference objects (e.g., the resource field in the preceding sample). Such objects should only be accessed when the method is called from the `IDisposable.Dispose` implementation (when the *disposing* parameter is equal to true). If the method is invoked from the finalizer (*disposing* is false), other objects should not be accessed. The reason is that objects are finalized in an unpredictable order and so they, or any of their dependencies, might already have been finalized.

Also, this section applies to classes with a base that does not already implement the Dispose Pattern. If you are inheriting from a class that already implements the pattern, simply override the `Dispose(bool)` method to provide additional resource cleanup logic.

✓ **DO** declare a protected virtual void `Dispose(bool disposing)` method to centralize all logic related to releasing unmanaged resources.

All resource cleanup should occur in this method. The method is called from both the finalizer and the `IDisposable.Dispose` method. The parameter will be false if being invoked from inside a finalizer. It should be used to ensure any code running during finalization is not accessing other finalizable objects. Details of implementing finalizers are described in the next section.

```

protected virtual void Dispose(bool disposing){
    if (disposing){
        if (resource != null) resource.Dispose();
    }
}

```

✓ **DO** implement the `IDisposable` interface by simply calling `Dispose(true)` followed by `GC.SuppressFinalize(this)`.

The call to `SuppressFinalize` should only occur if `Dispose(true)` executes successfully.

```

public void Dispose(){
    Dispose(true);
    GC.SuppressFinalize(this);
}

```

✗ **DO NOT** make the parameterless `Dispose` method virtual.

The `Dispose(bool)` method is the one that should be overridden by subclasses.

```

// bad design
public class DisposableResourceHolder : IDisposable {
    public virtual void Dispose() { ... }
    protected virtual void Dispose(bool disposing) { ... }
}

// good design
public class DisposableResourceHolder : IDisposable {
    public void Dispose() { ... }
    protected virtual void Dispose(bool disposing) { ... }
}

```

✗ **DO NOT** declare any overloads of the `Dispose` method other than `Dispose()` and `Dispose(bool)`.

`Dispose` should be considered a reserved word to help codify this pattern and prevent confusion among implementers, users, and compilers. Some languages might choose to automatically implement this pattern on certain types.

✓ **DO** allow the `Dispose(bool)` method to be called more than once. The method might choose to do nothing after the first call.

```

public class DisposableResourceHolder : IDisposable {

    bool disposed = false;

    protected virtual void Dispose(bool disposing){
        if(disposed) return;
        // cleanup
        ...
        disposed = true;
    }
}

```

✗ **AVOID** throwing an exception from within `Dispose(bool)` except under critical situations where the containing process has been corrupted (leaks, inconsistent shared state, etc.).

Users expect that a call to `Dispose` will not raise an exception.

If `Dispose` could raise an exception, further finally-block cleanup logic will not execute. To work around this, the user would need to wrap every call to `Dispose` (within the finally block!) in a try block, which leads to very complex cleanup handlers. If executing a `Dispose(bool disposing)` method, never throw an exception if disposing is false. Doing so will terminate the process if executing inside a finalizer context.

✓ **DO** throw an `ObjectDisposedException` from any member that cannot be used after the object has been disposed of.

```
public class DisposableResourceHolder : IDisposable {
    bool disposed = false;
    SafeHandle resource; // handle to a resource

    public void DoSomething(){
        if(disposed) throw new ObjectDisposedException(...);
        // now call some native methods using the resource
        ...
    }
    protected virtual void Dispose(bool disposing){
        if(disposed) return;
        // cleanup
        ...
        disposed = true;
    }
}
```

✓ **CONSIDER** providing method `Close()`, in addition to the `Dispose()`, if close is standard terminology in the area.

When doing so, it is important that you make the `Close` implementation identical to `Dispose` and consider implementing the `IDisposable.Dispose` method explicitly.

```
public class Stream : IDisposable {
    IDisposable.Dispose(){
        Close();
    }
    public void Close(){
        Dispose(true);
        GC.SuppressFinalize(this);
    }
}
```

Finalizable Types

Finalizable types are types that extend the Basic Dispose Pattern by overriding the finalizer and providing finalization code path in the `Dispose(bool)` method.

Finalizers are notoriously difficult to implement correctly, primarily because you cannot make certain (normally valid) assumptions about the state of the system during their execution. The following guidelines should be taken into careful consideration.

Note that some of the guidelines apply not just to the `Finalize` method, but to any code called from a finalizer. In the case of the Basic Dispose Pattern previously defined, this means logic that executes inside `Dispose(bool disposing)` when the *disposing* parameter is false.

If the base class already is finalizable and implements the Basic Dispose Pattern, you should not override `Finalize` again. You should instead just override the `Dispose(bool)` method to provide additional resource cleanup logic.

The following code shows an example of a finalizable type:

```
public class ComplexResourceHolder : IDisposable {

    private IntPtr buffer; // unmanaged memory buffer
    private SafeHandle resource; // disposable handle to a resource

    public ComplexResourceHolder(){
        this.buffer = ... // allocates memory
        this.resource = ... // allocates the resource
    }

    protected virtual void Dispose(bool disposing){
        ReleaseBuffer(buffer); // release unmanaged memory
        if (disposing){ // release other disposable objects
            if (resource != null) resource.Dispose();
        }
    }

    ~ComplexResourceHolder(){
        Dispose(false);
    }

    public void Dispose(){
        Dispose(true);
        GC.SuppressFinalize(this);
    }
}
```



```
}
```

X AVOID making types finalizable.

Carefully consider any case in which you think a finalizer is needed. There is a real cost associated with instances with finalizers, from both a performance and code complexity standpoint. Prefer using resource wrappers such as [SafeHandle](#) to encapsulate unmanaged resources where possible, in which case a finalizer becomes unnecessary because the wrapper is responsible for its own resource cleanup.

X DO NOT make value types finalizable.

Only reference types actually get finalized by the CLR, and thus any attempt to place a finalizer on a value type will be ignored. The C# and C++ compilers enforce this rule.

✓ **DO** make a type finalizable if the type is responsible for releasing an unmanaged resource that does not have its own finalizer.

When implementing the finalizer, simply call `Dispose(false)` and place all resource cleanup logic inside the `Dispose(bool disposing)` method.

```
public class ComplexResourceHolder : IDisposable {  
    ~ComplexResourceHolder() {  
        Dispose(false);  
    }  
  
    protected virtual void Dispose(bool disposing) {  
        ...  
    }  
}
```

✓ **DO** implement the Basic Dispose Pattern on every finalizable type.

This gives users of the type a means to explicitly perform deterministic cleanup of those same resources for which the finalizer is responsible.

X DO NOT access any finalizable objects in the finalizer code path, because there is significant risk that they will have already been finalized.

For example, a finalizable object A that has a reference to another finalizable object B cannot reliably use B in A's finalizer, or vice versa. Finalizers are called in a random order (short of a weak ordering guarantee for critical finalization).

Also, be aware that objects stored in static variables will get collected at certain points during an application domain unload or while exiting the process. Accessing a static variable that refers to a finalizable object (or calling a static method that might use values stored in static variables) might not be safe if [Environment.HasShutdownStarted](#) returns true.

✓ **DO** make your `Finalize` method protected.

C#, C++, and VB.NET developers do not need to worry about this, because the compilers help to enforce this guideline.

X DO NOT let exceptions escape from the finalizer logic, except for system-critical failures.

If an exception is thrown from a finalizer, the CLR will shut down the entire process (as of .NET Framework version 2.0), preventing other finalizers from executing and resources from being released in a controlled manner.

✓ **CONSIDER** creating and using a critical finalizable object (a type with a type hierarchy that contains [CriticalFinalizerObject](#)) for situations in which a finalizer absolutely must execute even in the face of forced application domain unloads and thread aborts.

Portions © 2005, 2009 Microsoft Corporation. All rights reserved.

Reprinted by permission of Pearson Education, Inc. from *Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries, 2nd Edition* by Krzysztof Cwalina and Brad Abrams, published Oct 22, 2008 by Addison-Wesley Professional as part of the Microsoft Windows Development Series.

▲ See Also

Reference

[IDisposable.Dispose](#)
[Object.Finalize](#)

Other Resources

[Framework Design Guidelines](#)
[Common Design Patterns](#)
[Garbage Collection](#)

